

Principles of Secure Processor Architecture Design



Slides and information available at:

<http://caslab.csl.yale.edu/tutorials/hpca2019/>

Principles of Secure Processor Architecture Design



Jakub Szefer
Assistant Professor
Dept. of Electrical Engineering
Yale University

HPCA 2019 – February 17th, 2019



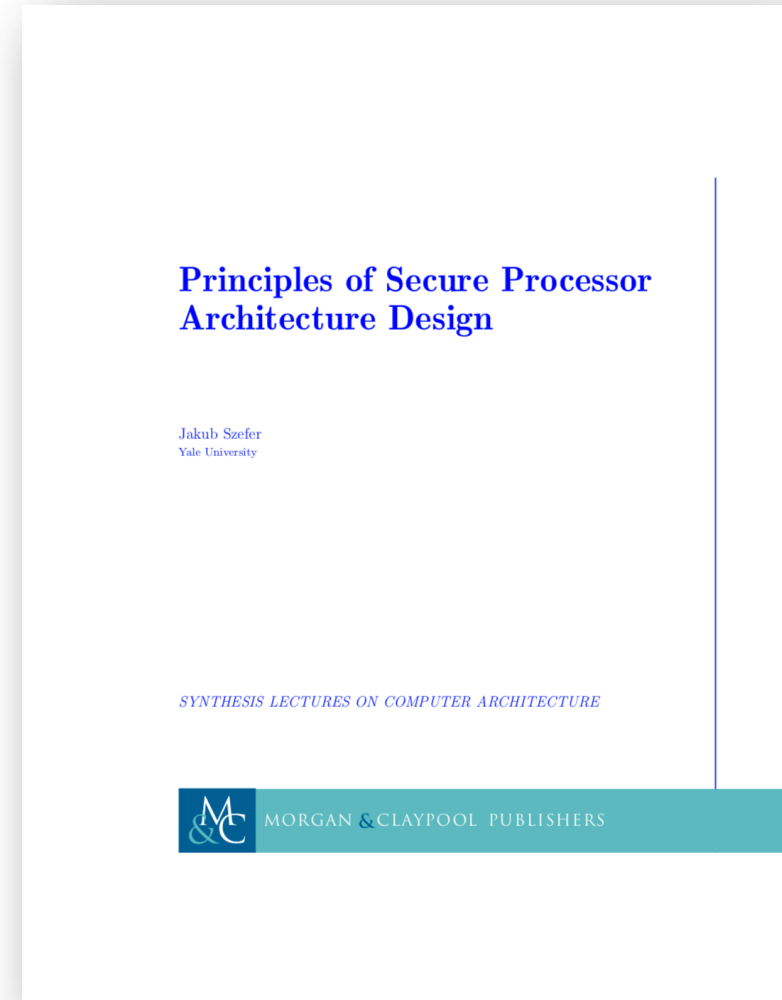
13:00 – 13:30	Secure Processor Architectures
13:30 – 14:00	Trusted Execution Environments
14:00 – 14:10	Break
14:10 – 14:30	Hardware Roots of Trust
14:30 – 14:50	Memory Protection
14:50 – 15:00	Multiprocessor and Many-core Protections
15:00 – 15:10	Break
15:10 – 15:50	Side-Channels Threats and Protections including Speculative Execution Threats
15:50 – 16:20	Principles of Secure Processor Architecture Design
16:20 – 16:30	Ending

The Book



Jakub Szefer, "Principles of Secure Processor Architecture Design," in Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, October 2018.

<http://caslab.csl.yale.edu/books/>





Secure Processor Architectures

Trusted Execution Environments

Hardware Roots of Trust

Memory Protection

Multiprocessor and Many-core Protections

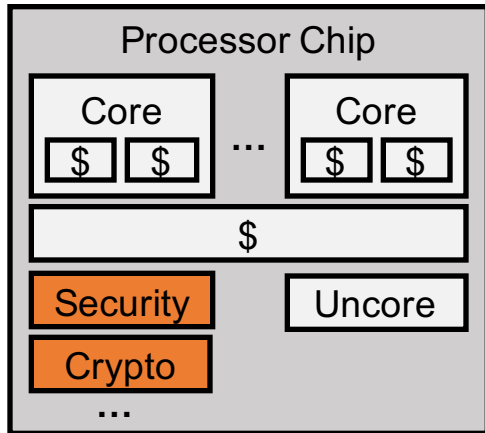
Side-Channels Threats and Protections

Principles of Secure Processor Architecture Design

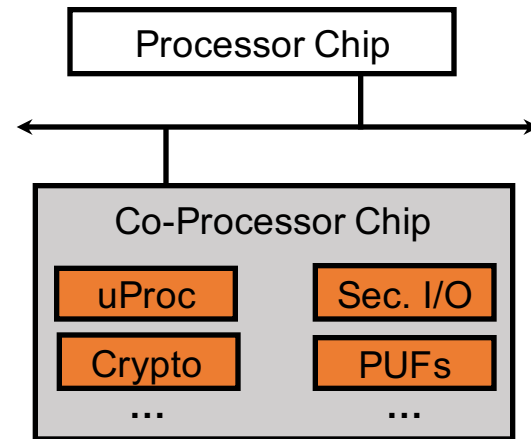
Types of Security Related Architectures



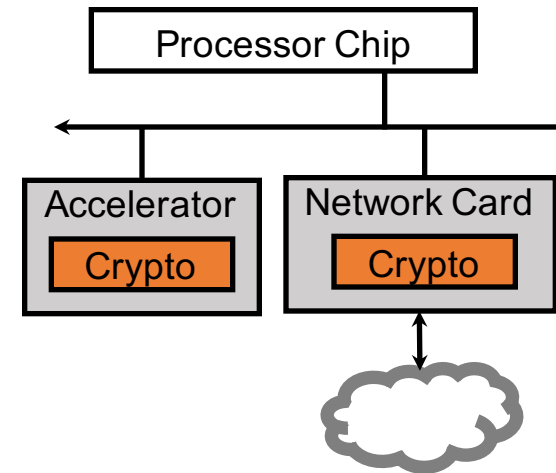
Secure Processor Architecture



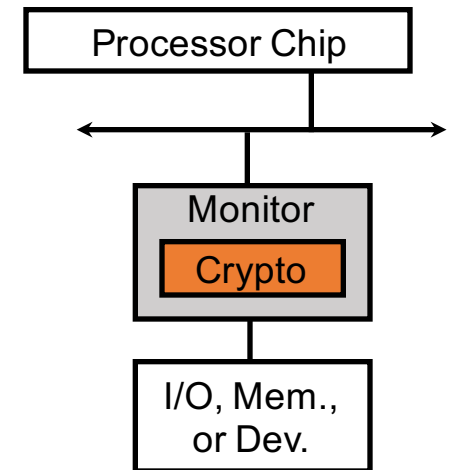
Secure Co-Processor or HSM Architecture



Cryptographic Accelerators



Security Monitor



Brief History of Secure Processor Architectures



Starting in late 1990s or early 2000s, academics have shown increased interest in secure processor architectures:

XOM (2000), AEGIS (2003), Secret-Protecting (2005), Bastion (2010),
NoHype (2010), HyperWall (2012), CHERI (2014), Sanctum (2016),
Keystone (about 2017), MI6 (2018)

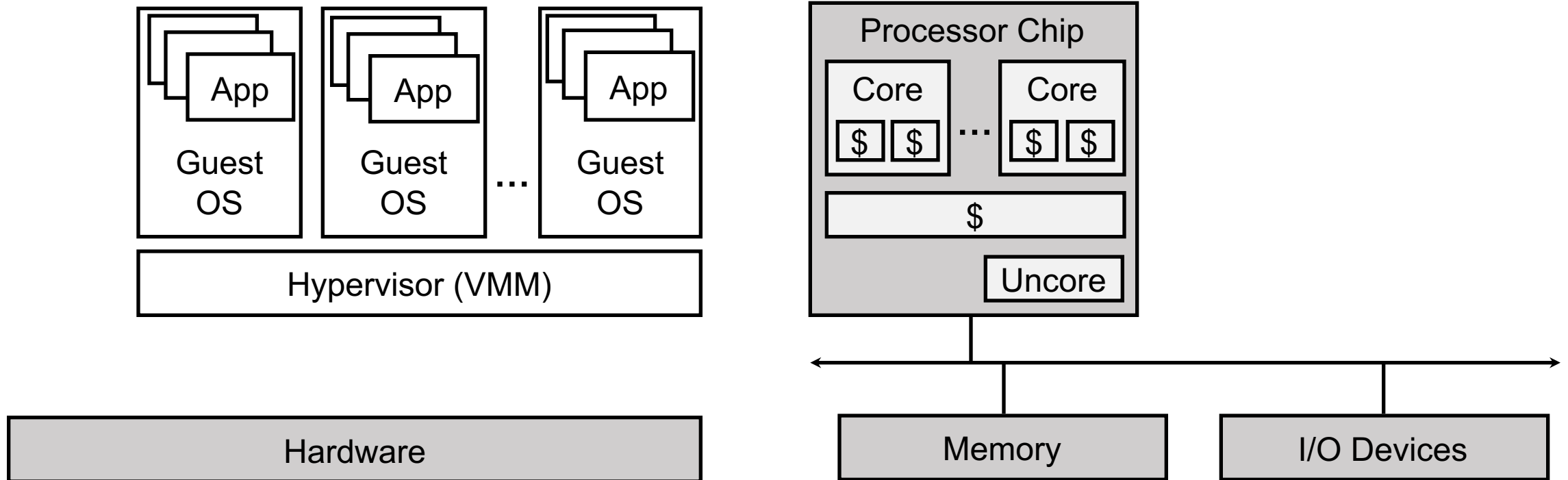
Commercial processor architectures have also included security features:

LPAR in IBM mainframes (1970s), Security Processor Vault in Cell Broadband Engine (2000s),
ARM TrustZone (2000s), Intel TXT & TPM module (2000s), Intel SGX (mid 2010s),
AMD SEV (late 2010s)

Baseline (Unsecure) Processor Architecture



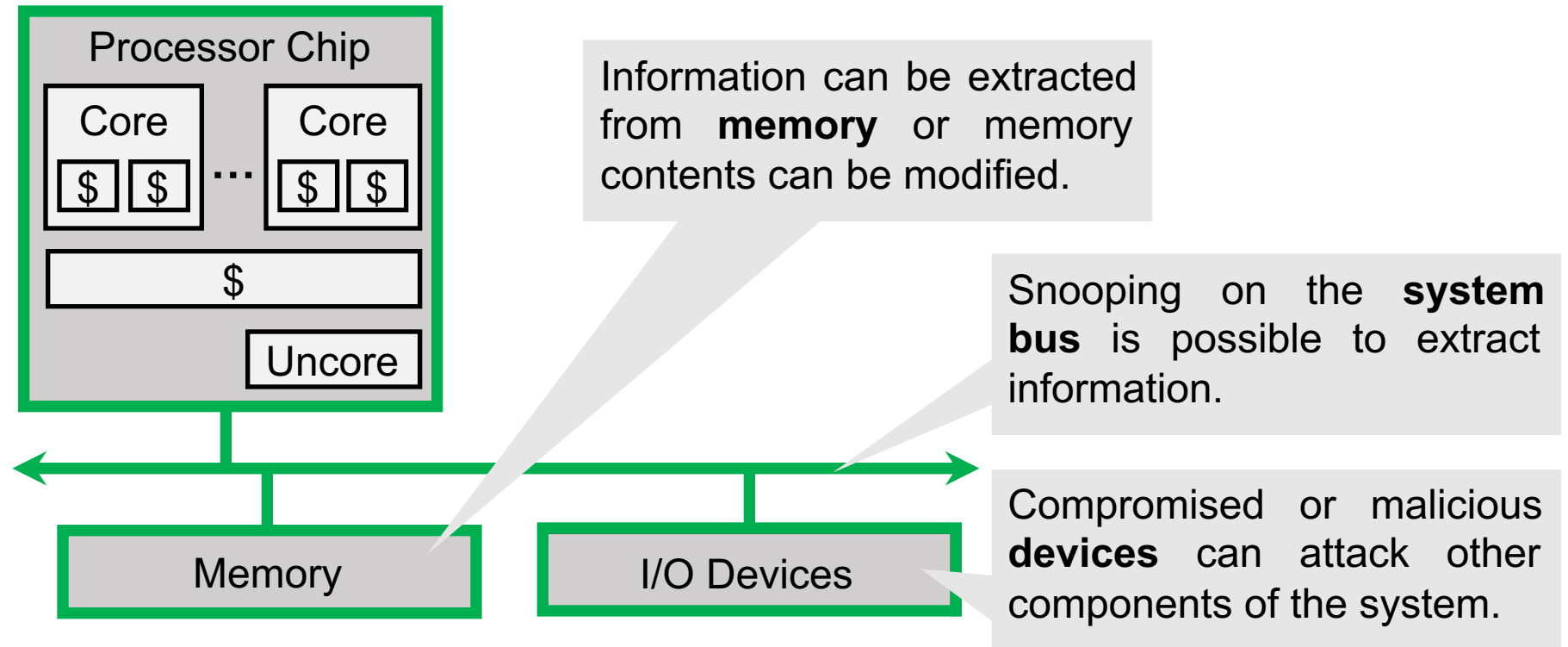
A simplified view of a processor and the software stack in a general-purpose computer:



Baseline (Unsecure) Processor Hardware



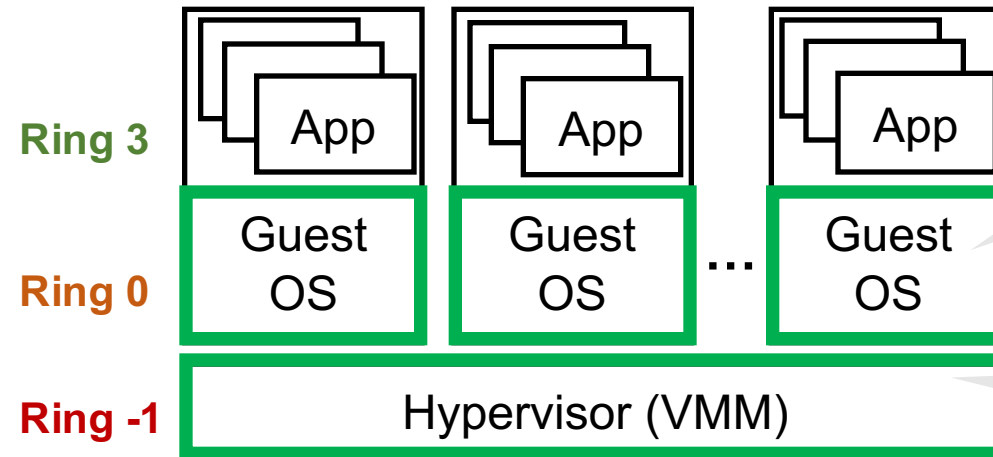
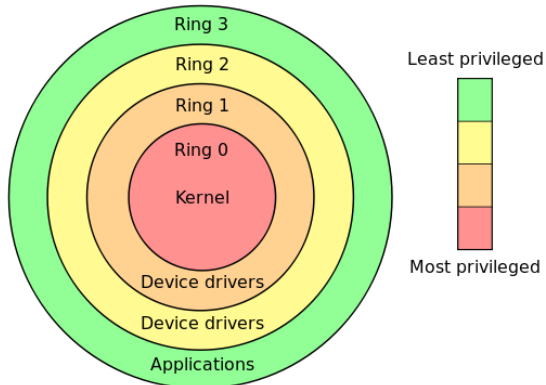
Typical computer system with no secure components nor secure processor architectures considers all the components as trusted:



Baseline (Unsecure) Processor Software



Typical computer system uses ring-based protection scheme, which gives most privileges (and most trust) to the lowest levels of the system software:



Compromised or malicious **OS** can attack all the applications in the system.

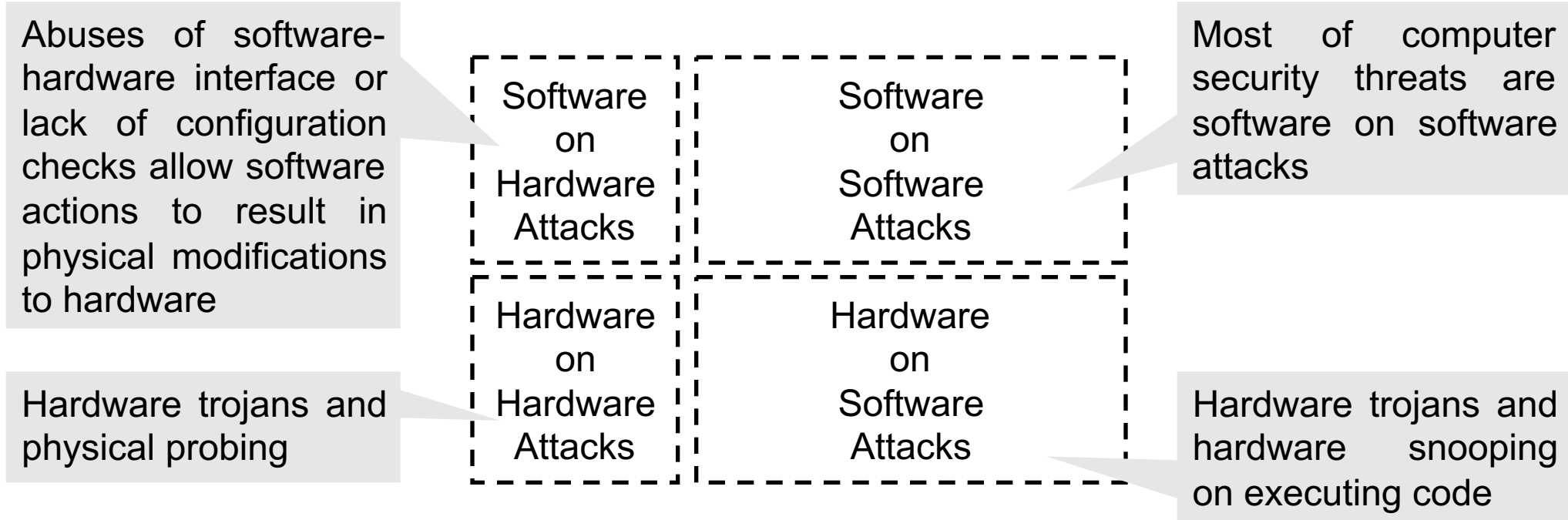
Compromised or malicious **Hypervisor** can attack all the OSes in the system.

Image:
https://commons.wikimedia.org/wiki/File:Priv_rings.svg

Potential Attack Vectors



Hardware and software can be attacked through numerous attack vectors:



Software on Software Attacks



Most computer security attacks are software attacks, typically targeting other software running on the same computer, or targeting the OS or Hypervisor.

E.g., Return-Oriented Programming (ROP):

- Does not require loading attackers' code onto victim machine
- Requires modification of the call stack
- Uses “gadgets” present already on computer as part of some software or library

Sample defense: Address Space Layout Randomization (ASLR)

General software attack types:

- Control flow modification
- Data modification

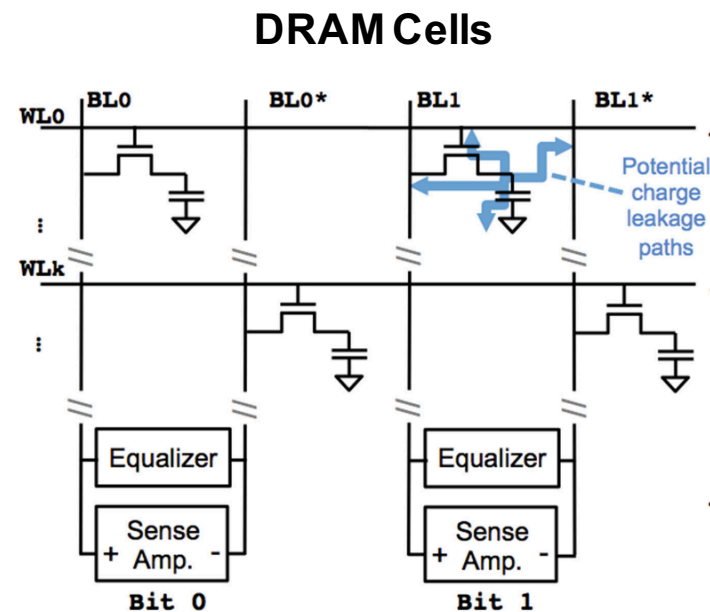
Software on Hardware Attacks



Software can be leveraged to attempt to modify physical properties of the hardware.

Rowhammer attack:

- Repeated accesses to DRAM rows can cause bits to flip *in adjacent* DRAM rows
- Leverage usual load or store instructions, but at a very high rate
- Can be triggered by DMA from devices as well
- Use attack to change protection bits in a page table, etc.



DRAM Bank



Images:
(left) Xiong, et al., "Run-Time Accessible DRAM PUFs in Commodity Devices"
(right) The Hacker News

Software on Hardware Attacks

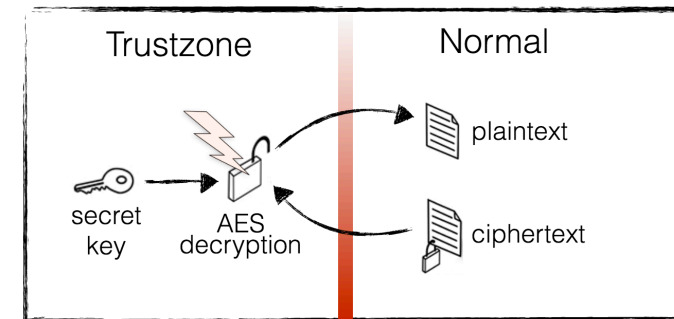


Software can be leveraged to attempt to modify physical properties of the hardware.

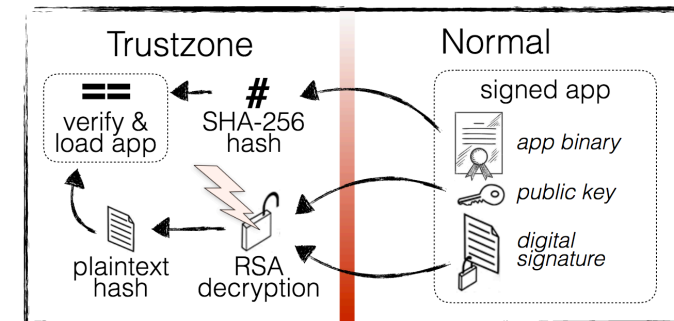
CLKScrew attack:

- Abuses Dynamic Voltage and Frequency Scaling (DVFS) features
- Adjust configuration beyond normally allowed operating points
- Inject faults into system
 - Confidentiality – inject faults and use differential fault attacks to get secret key
 - Integrity – inject faults to get data verification to pass
 - Availability – crash system with too many faults

Confidentiality Attack



Integrity Attack



Images:

USENIX slides for "CLKScrew: Exposing the Perils of Security-Oblivious Energy Management"

Hardware on Hardware Attacks



Require hardware modification (hardware trojans) or physical proximity

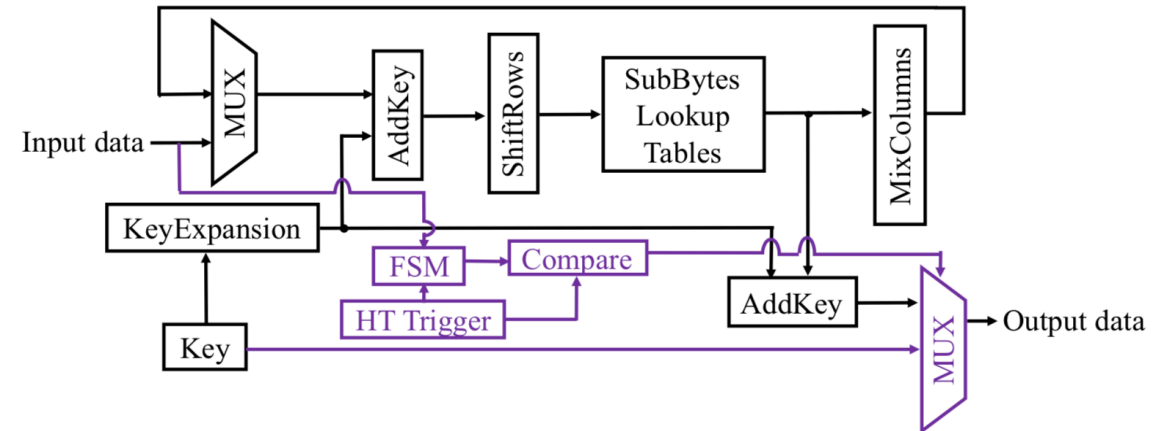
Hardware trojans:

- Hardware is modified to add hidden functionality
 - In source code, 3rd party IP modules, malicious CAD tools, at the foundry, physically after manufacturing
- Modify hardware behavior or extract secrets

Other attack categories:

- Physical extraction after attack (e.g. probing)
- Side channels: power, EM, thermal
- Fault injection (using dedicated physical equipment)

Sample Hardware Trojan (HT) in AES



Hardware on Software Attacks



Leverage hardware modification to change behavior of the software or extract some secrets.

Exfiltration of software secrets:

- Leverage hardware's access to registers and memory to read out data
- Example attacks with snooping on memory bus

Sample defense: don't make all hardware trusted, e.g. use memory encryption

Attacking Hardware without Physical Access



Possibilities for hardware attacks with dedicated tools and lots of money are infinite.

However, **software on hardware attacks requiring no physical access** are possible today.

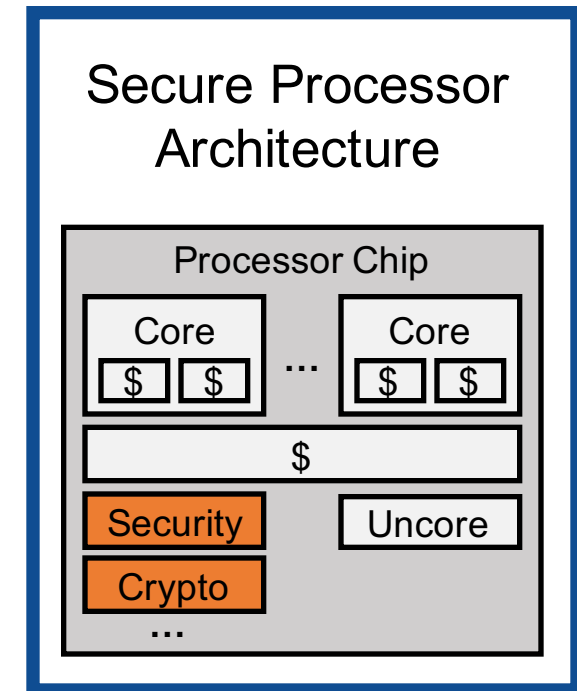
- Rowhammer Repeated accesses to DRAM rows can cause bits to flip in adjacent DRAM rows, e.g. to change protection bits in a page table.
- CLKScrew Abusing Dynamic Voltage and Frequency Scaling (DVFS) features can allow attacker to introduce faults into a system.
- ...
- Meltdown Out-of-order execution and incorrect checking of protection bits + cache side channel attacks can leak information about protected memory contents.
- Spectre Speculative execution + cache side channel attacks can be used to extract data from an application.
- ...

Protecting from Software and Hardware Attacks



Secure Processor Architectures add new hardware and software features to provide **Trusted Execution Environments (TEEs)** wherein software executes protected from some of the software and hardware threats.

- Enhance general-purpose processor with new protection features
- Provide new or alternate privilege levels
- Utilize software and hardware changes
- Facilitate attestation of the protected software



New Privilege Levels



Modern computer systems define protections in terms of **privilege level** or protection rings, new privilege levels are defined to provide added protections.

- Ring 3** Application code, least privileged.
- Rings 2 and 1** Device drivers and other semi-privileged code, although rarely used.
- Ring 0** Operating system kernel.
- Ring -1** Hypervisor or virtual machine monitor (VMM), most privileged mode that a typical system administrator has access to.
- Ring -2** System management mode (SMM), typically locked down by processor manufacturer
- Ring -3** Platform management engine, retroactively named “ring -3”, actually runs on a separate management processor.

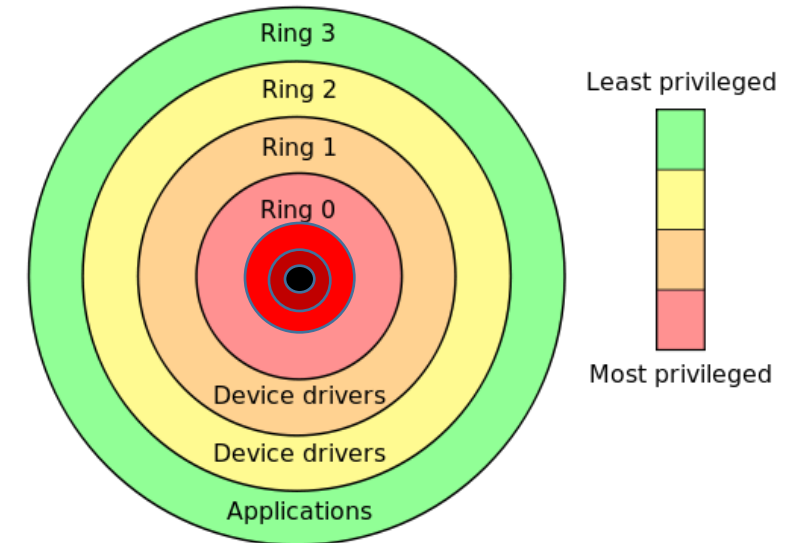


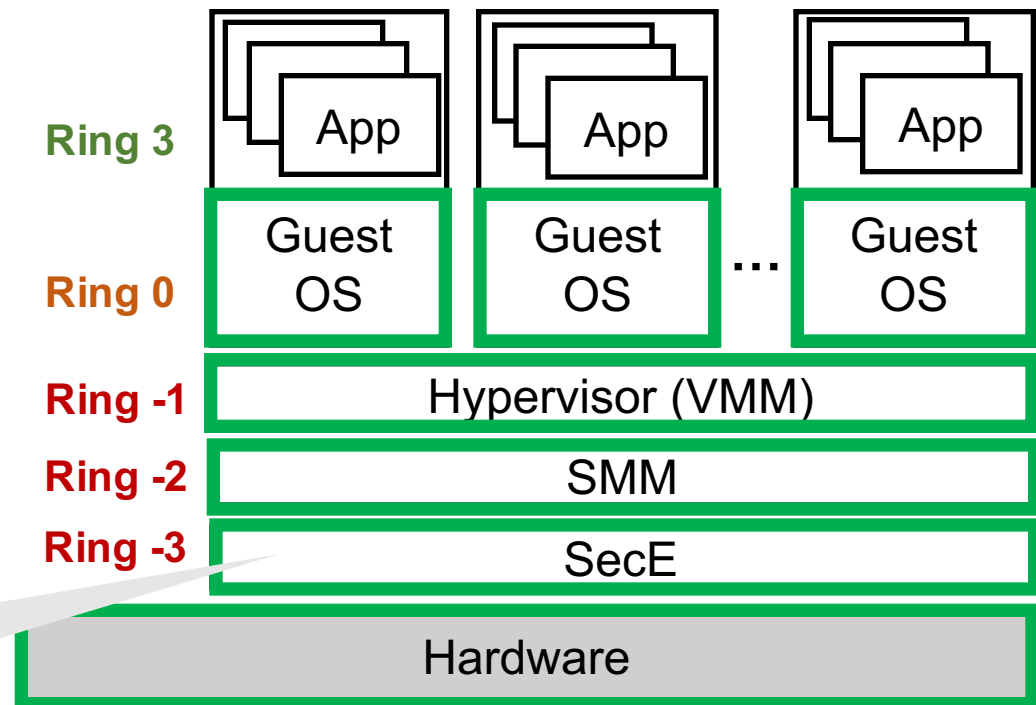
Image:
https://commons.wikimedia.org/wiki/File:Priv_rings.svg

Extend Linear Trust with New Protection Levels



The hardware is most privileged as it is the lowest level in the system.

- There is a linear relationship between protection ring and privilege (lower ring is more privileged)
- Each component **trusts** all the software “below” it



Security Engine (SecE) can be something like Intel's ME or AMD's PSP.

Add Horizontal Privilege Separation



New privileges can be made orthogonal to existing protection rings.

- E.g. ARM's TrustZone's "normal" and "secure" worlds
- Need privilege level (ring number) and normal / secure privilege

Security levels from a lattice:

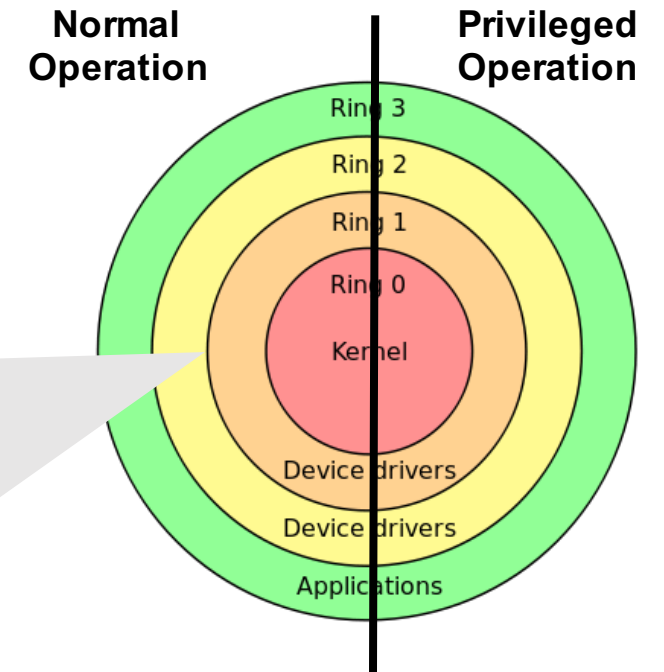
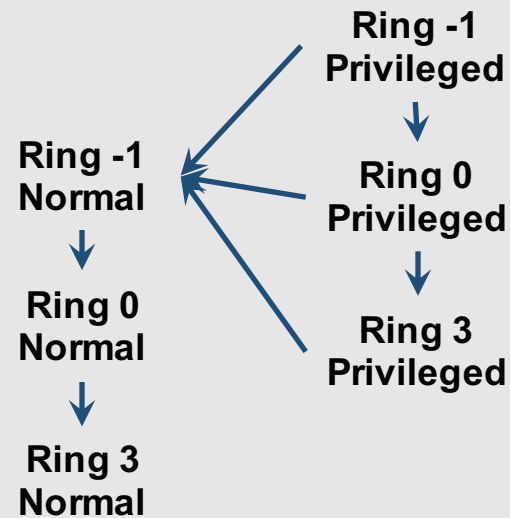
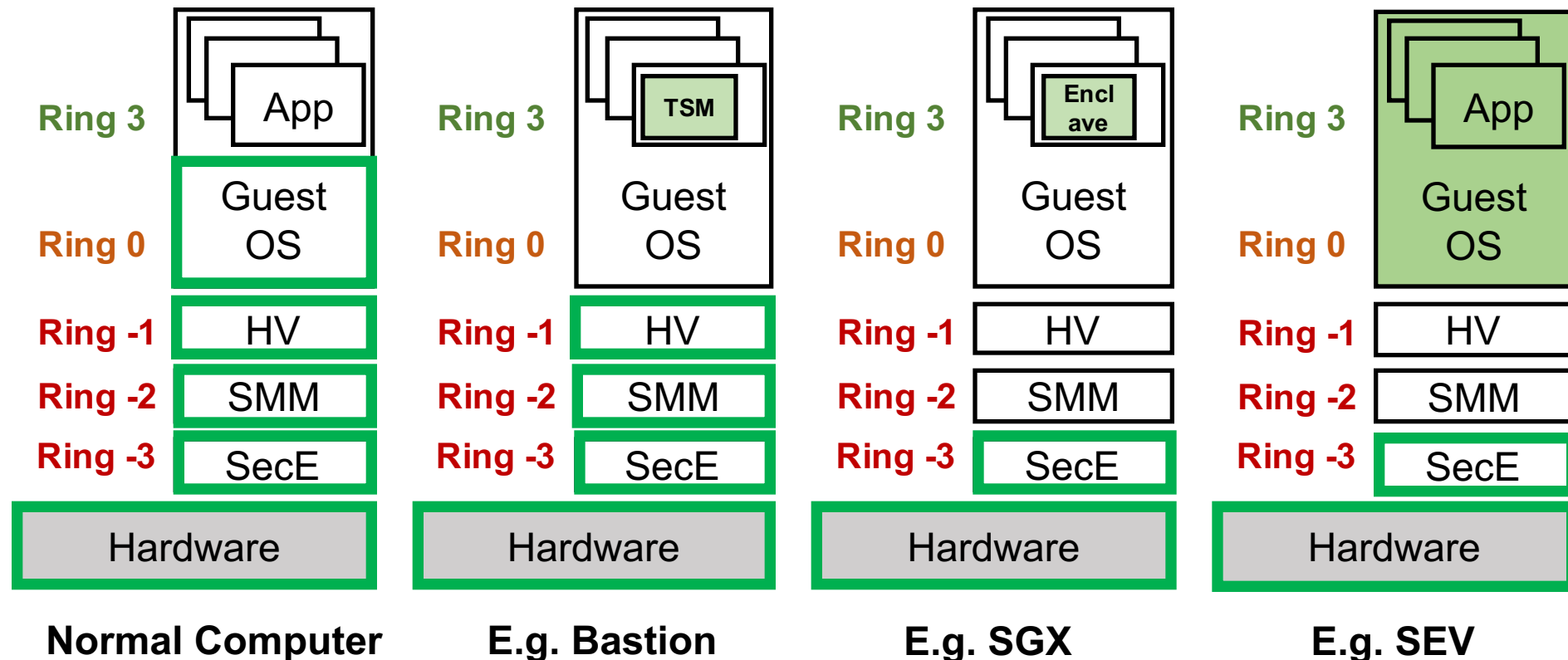


Image:
https://commons.wikimedia.org/wiki/File:Priv_rings.svg

Breaking Linear Hierarchy of Protection Rings



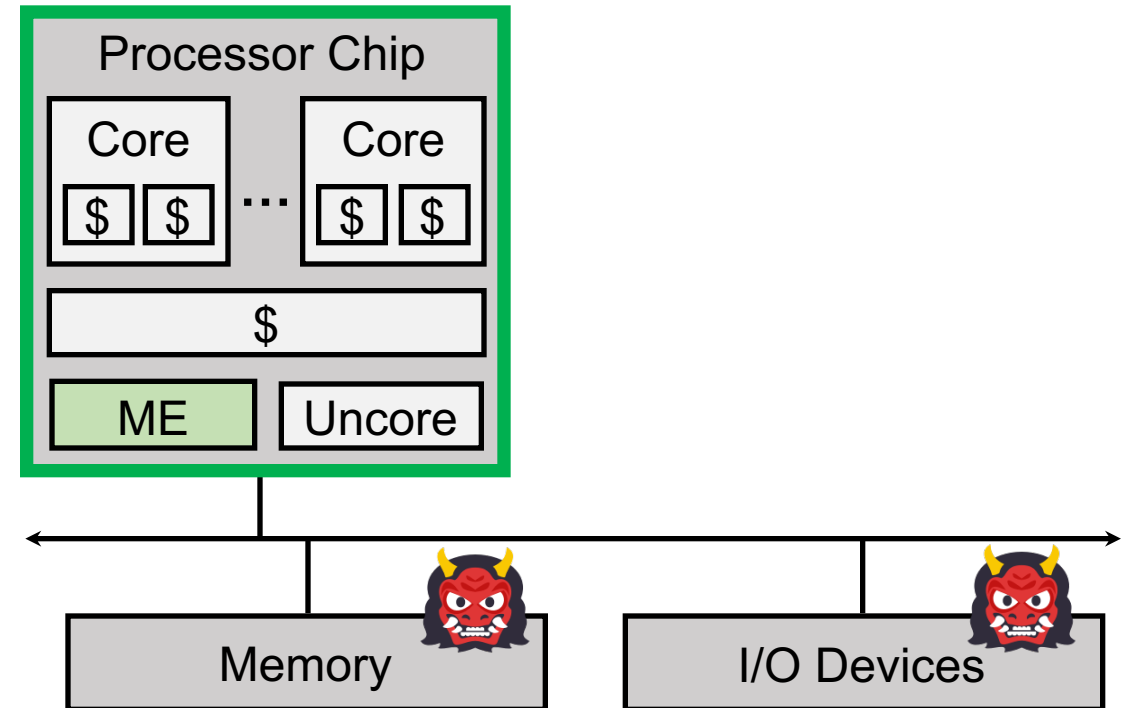
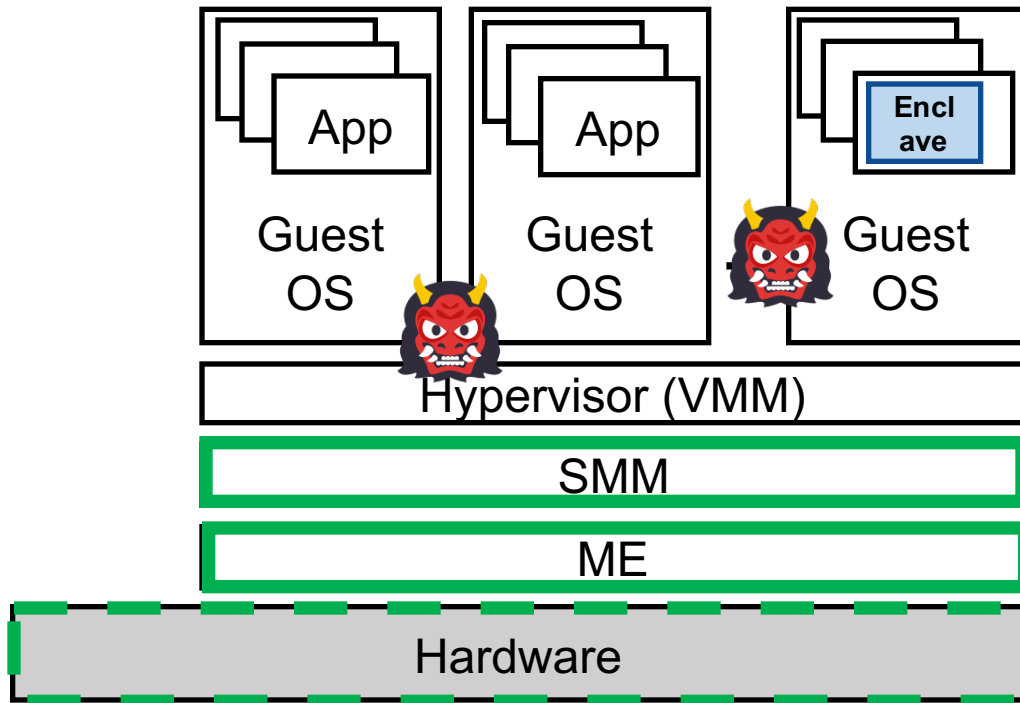
Examples of architectures that do and don't have a linear relationship between privileges and protection ring level:



Example Secure Architecture: Intel SGX



Simplified schematic of Intel SGX architecture and the protected enclaves.

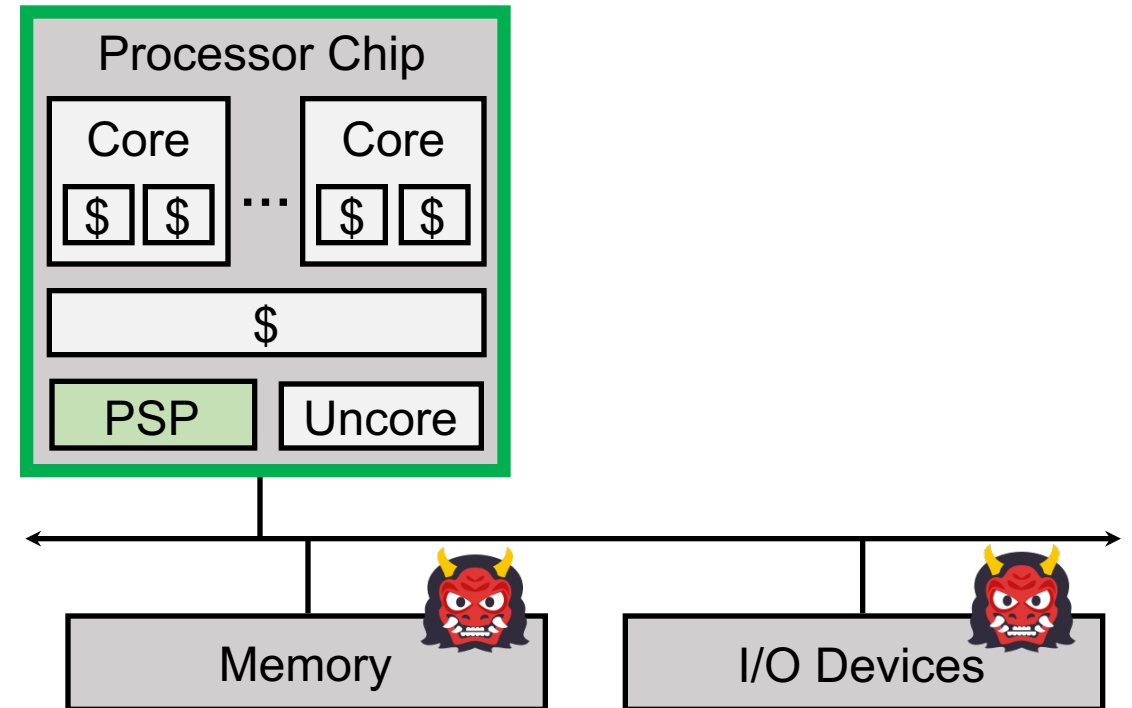
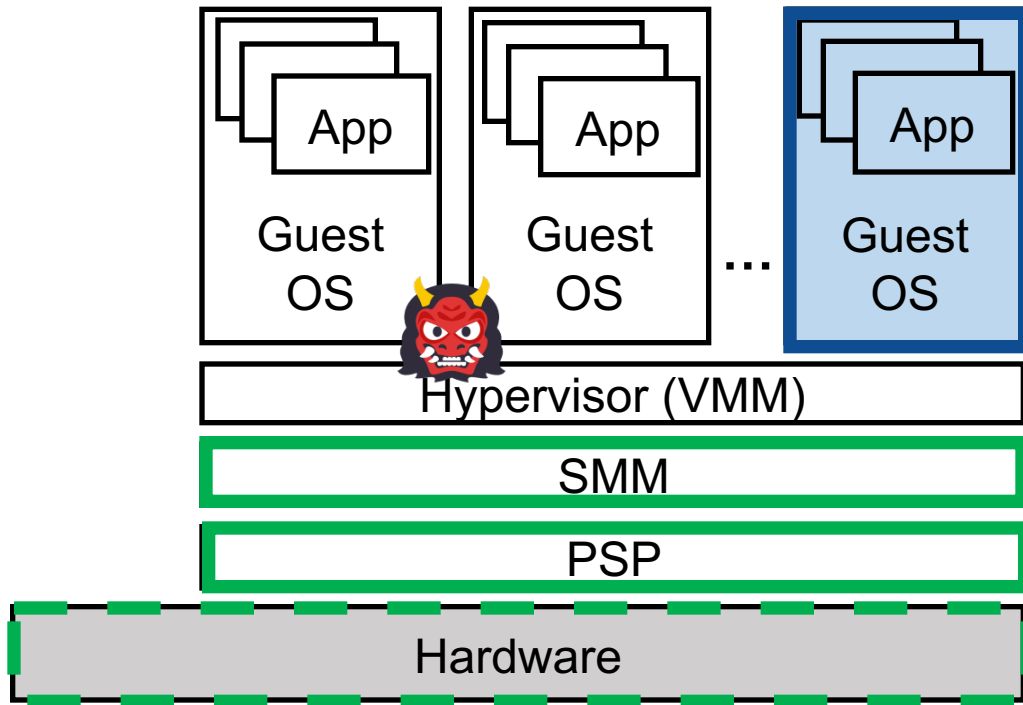


Emoji Image:
<https://www.emojione.com/emoji/1f479>

Example Secure Architecture: AMD SEV



Simplified schematic of AMD SEV architecture and the protected Virtual Machines.

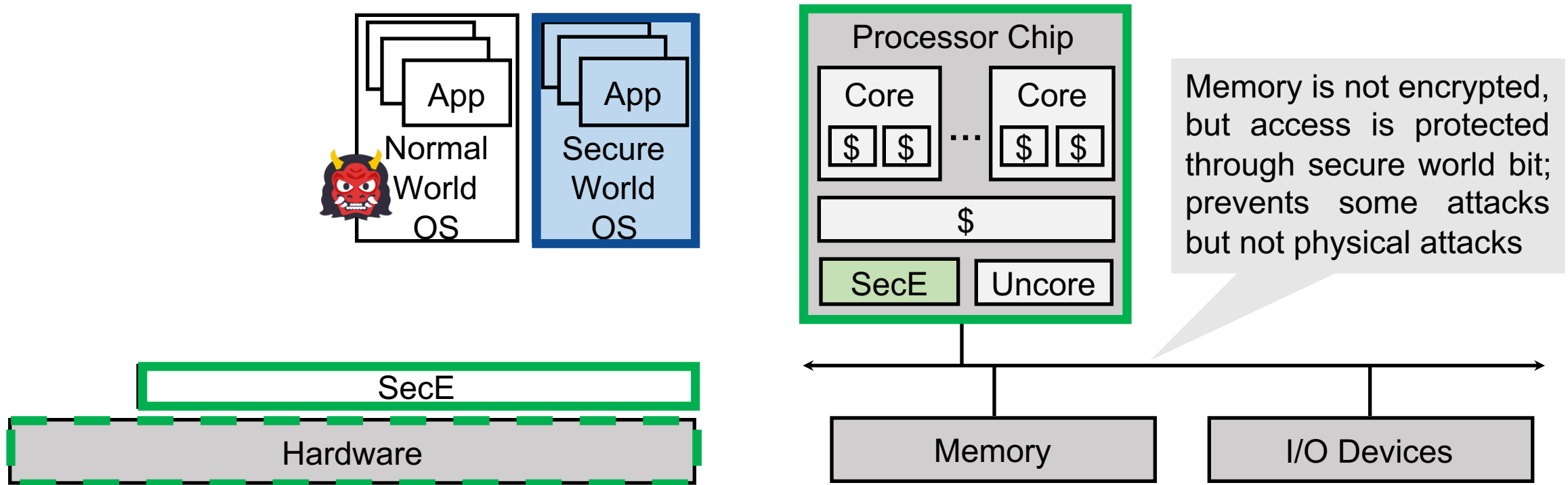


Emoji Image:
<https://www.emojione.com/emoji/1f479>

Example Secure Architecture: ARM TrustZone



Simplified schematic of ARM TrustZone architecture and the normal and protected worlds.

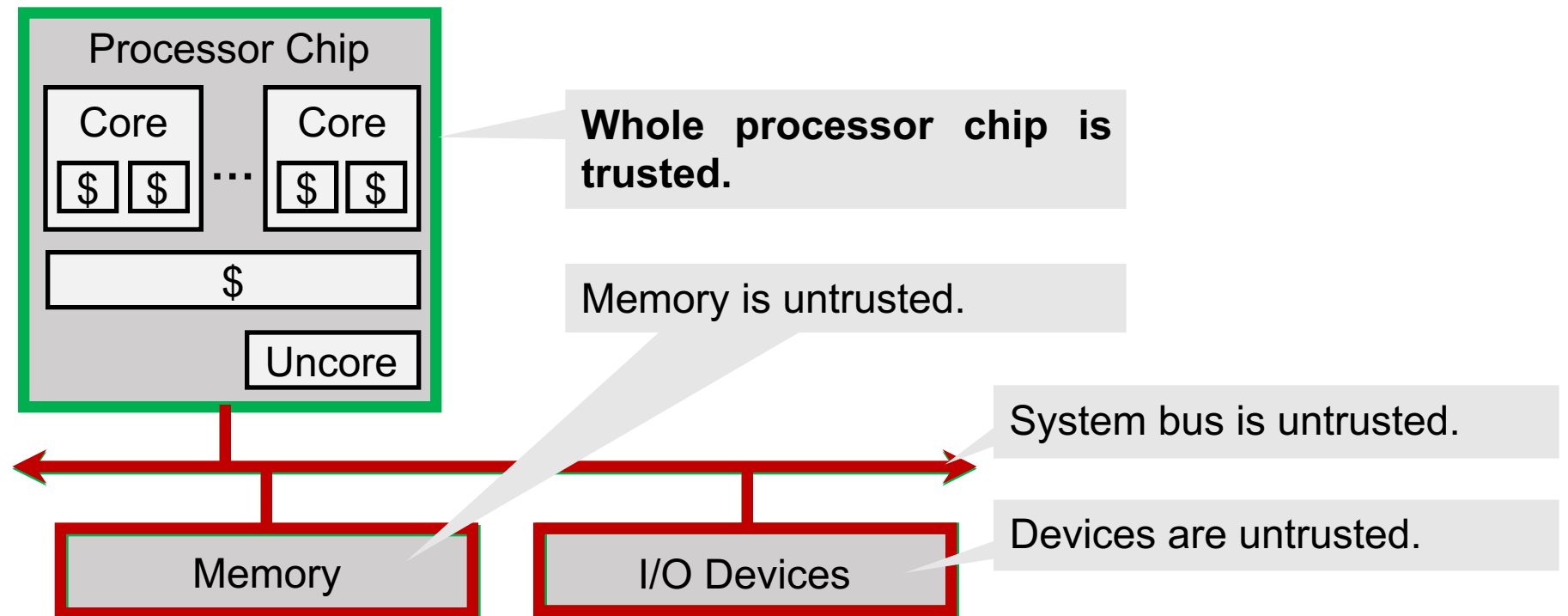


Emoji Image:
<https://www.emojione.com/emoji/1f479>

Trusted Processor Chip Assumption



Key to most secure processor architecture designs is the **trusted processor chip assumption**.



Trusted Computing Base



Trusted Computing Base, or **TCB**, is the sum total of all the hardware and software which work together to realize the protections offered by the system.

- TCB is trusted
- TCB may not be trustworthy, if is not verified or is not bug free

TCB contains:

- All trusted hardware – typically the processor chip
- All trusted software – some software levels may be untrusted (e.g. OS in SGX)

Small TCB Assumption



To prevent TCB problems, TCB should be small; it is assumed that a smaller hardware and software TCB implies better security.

The **small TCB assumption** is derived from:

- Less software code means it can be audited and verified
- Less hardware code means it can be audited and verified

Limitations in today's security verification tools necessitate the small TCB assumption.

- Difficult to verify large code bases (both hardware and software)
- Hard to define all security policies for large, complex systems

Open TCB Assumption



Kerckhoffs's Principle from cryptography can be applied to secure architectures:

- Operation of the TCB should be publicly known and should have no secrets
- Only secrets are the cryptographic keys
- Prevent security-by-obscurity

Spectre, Meltdown, Foreshadow and other attacks could be attributed to security-by-obscurity as well. Microarchitectural operation of the processor is not (clearly) publicly known.

Today's Limitations of Secure Architectures



Threats which are outside the scope of secure processor architectures:

- Bugs or Vulnerabilities in the TCB
- Hardware Trojans and Supply Chain Attacks
- Physical Probing and Invasive Attacks

TCB hardware and software is prone to bugs just like any hardware and software.

Modifications to the processor after the design phase can be sources of attacks.

At runtime hardware can be probed to extract information from the physical realization of the chip.

Threats which are underestimated when designing secure processor architectures:

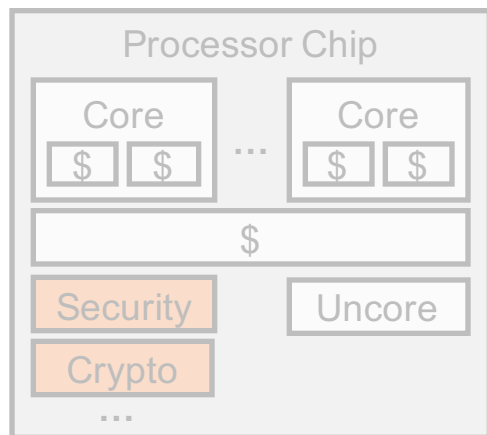
- Side Channel Attacks

Information can leak through timing, power, or electromagnetic emanations from the implementation

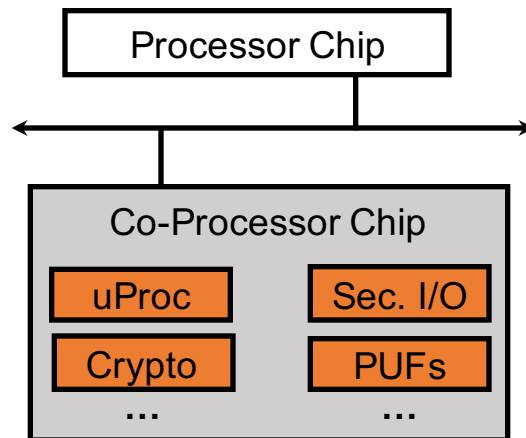
Other Security Related Architectures



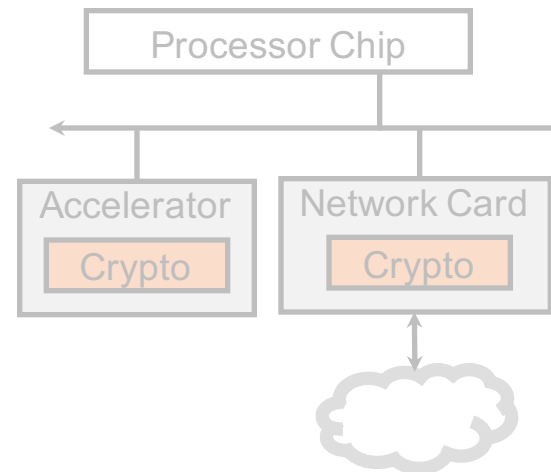
Secure Processor Architecture



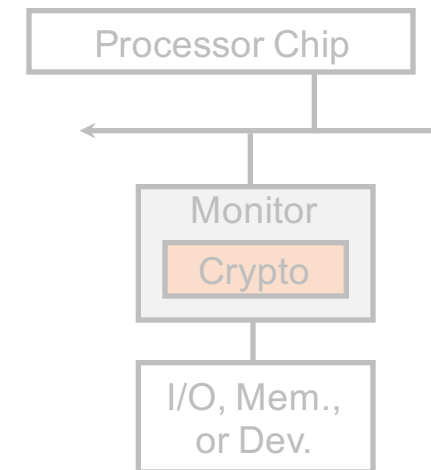
Secure Co-Processor or HSM Architecture



Cryptographic Accelerators



Security Monitor



Secure Co-Processors or HSMs



- Hardware security modules (HSMs) are dedicated devices for performing cryptographic operations or running secure code
 - Can be attached to the system bus such as PCIe, e.g. IBM cards
 - Can be integrated into SoC design and attached to AXI or similar bus
- Discrete HSMs typically have tamper resistant and tamper evident coatings, or have battery for backup power
 - Secure co-processors on SoC may lack battery backup, may have lesser physical tamper detection



Images:
<https://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml>

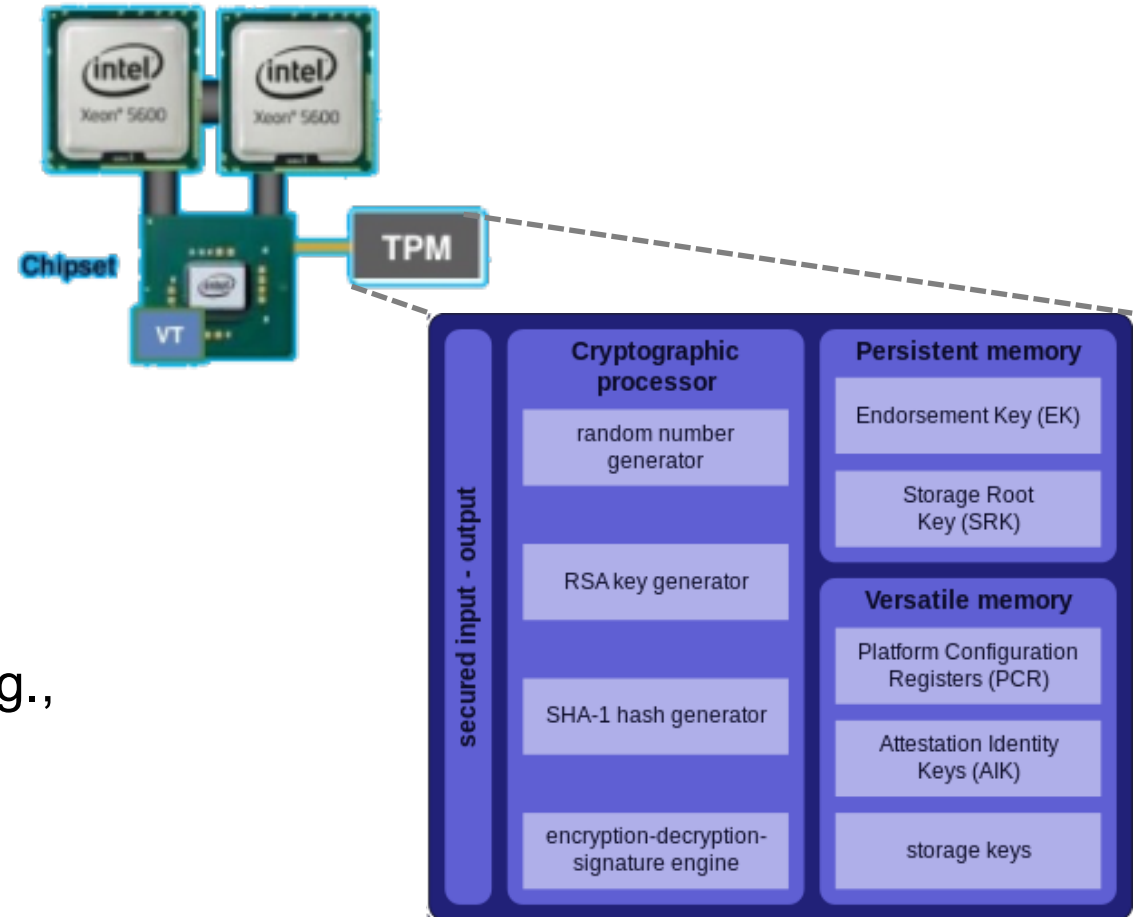
Intel – TPM and TXT



Co-processor developed by IBM and others

- Model: TPM v1.2
- Co-processor attaching to chipset
- Some features advertised by the vendor:
 - crypto engine

The Trusted Platform Module (TPM) is often integrated with other processor security features, e.g., Intel's Trusted Execution Technology (TXT)



Images:
<https://upload.wikimedia.org/wikipedia/commons/thumb/b/be/TPM.svg>
<https://intelsgx.blogspot.com/2016/05/intel-sgx-vs-txt.html>

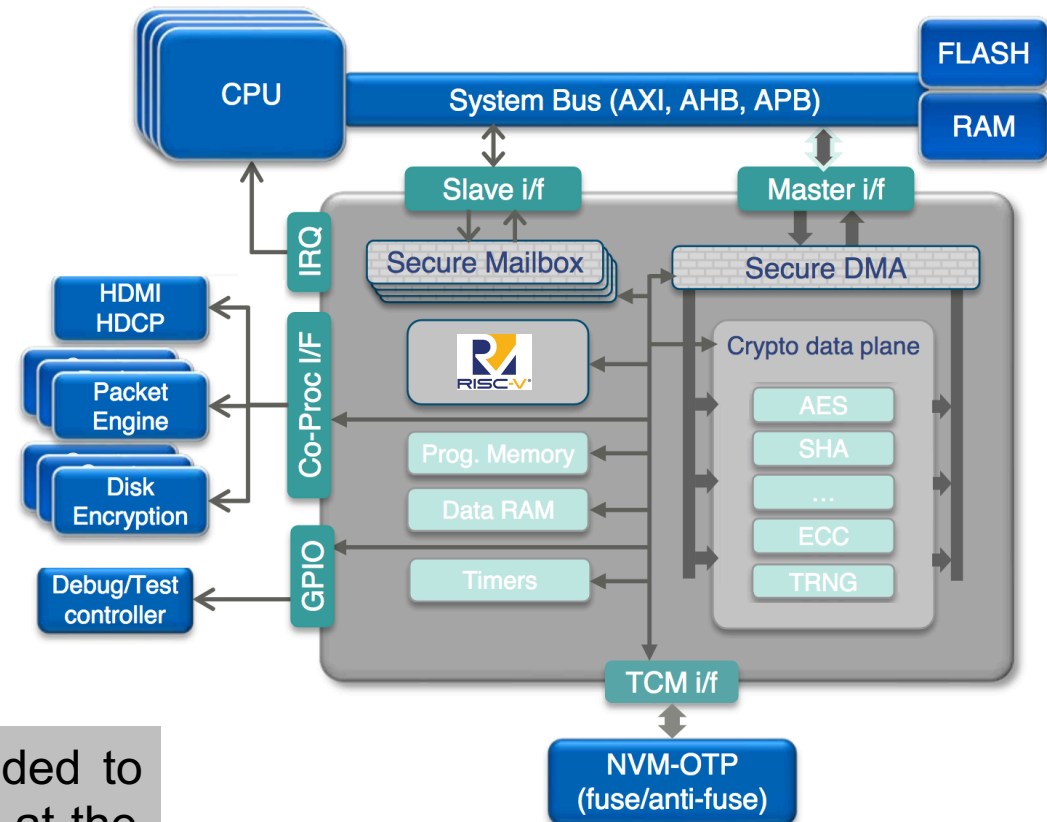
Inside Secure – Programmable Root-of-Trust



Co-processor IP developed by Inside Secure

- Model: Programmable Root-of-Trust
- Co-processor attaching to standard buses
- Some features advertised by the vendor:
 - crypto accelerators
 - can be synthesized with SypherMedia Library (SML) circuit camouflage technology and anti-reverse engineering
 - side channel protection
 - anti-tampering
 - secure debug

Camouflaging can be added to almost any design as it is at the level of logic gates.



Thanks to **Benoît De Dinechin** for suggesting to add SoC security solutions.

http://www.design-reuse-embedded.com/webinar/ip-soc-china-2017/slides/Inside%20Secure%20IPSOC_China_2017_Sept_V02_Public.pdf

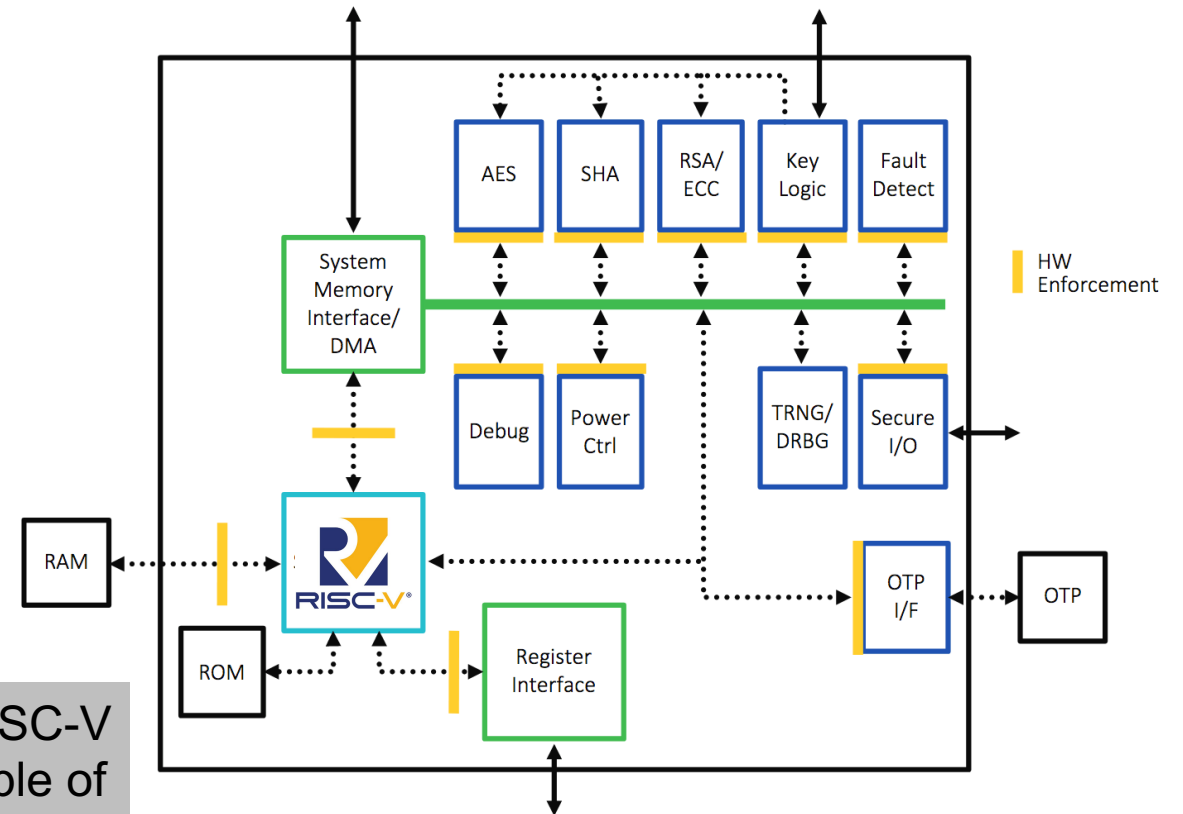
Rambus (Cryptography Research, Inc.) – CryptoManager



Co-processor IP developed by Cryptography Research, Inc. (CRI)

- Model: CryptoManager Root of Trust RT630
- Independent hardware security block
- Some features advertised by vendor:
 - crypto accelerators and DPA resistant crypto
 - “entropic array logic” to thwart reverse engineering
 - canary logic for protection against glitching and overclocking
 - not susceptible to Spectre and Meltdown
 - secure memories
 - anti-temper
 - secure debug

Probably due to simple RISC-V without speculation. Example of trading performance for security.



Thanks to **Benoît De Dinechin** for suggesting to add SoC security solutions.

<https://info.rambus.com/hubfs/rambus.com/Gated-Content/Cryptography/CryptoManager-Root-of-Trust-RT630-Product-Brief.pdf> Image:

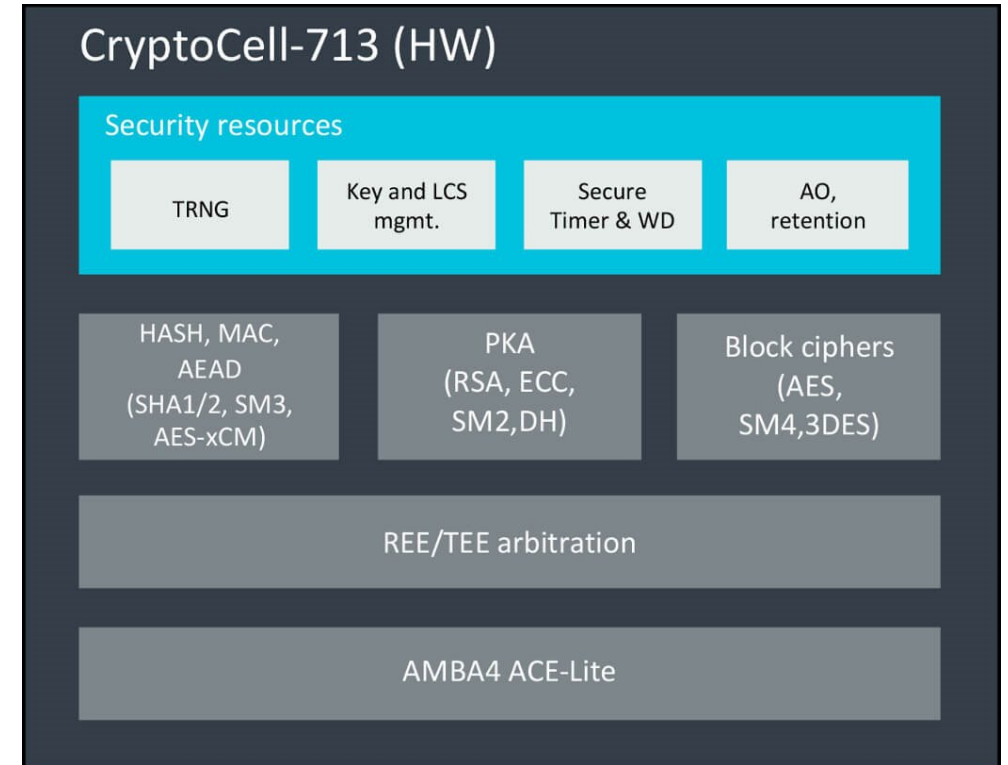
ARM – CryptoCell



Co-processor IP developed by Inside Secure

- Model: CryptoCell-713
- Co-processor attaching to ARM's buses
- Some features advertised by the vendor:
 - crypto accelerators
 - side channel protection
 - supports Chinese crypto algorithms: SM2 (public key alg. based on elliptic curves), SM3 (hash function), and SM4 (block cipher)

Example that cryptographic accelerators can be for different types of crypto.



Thanks to **Benoît De Dinechin** for suggesting to add SoC security solutions.

Image: <https://community.arm.com/processors/b/blog/posts/new-cryptocell-security-ip-announced>

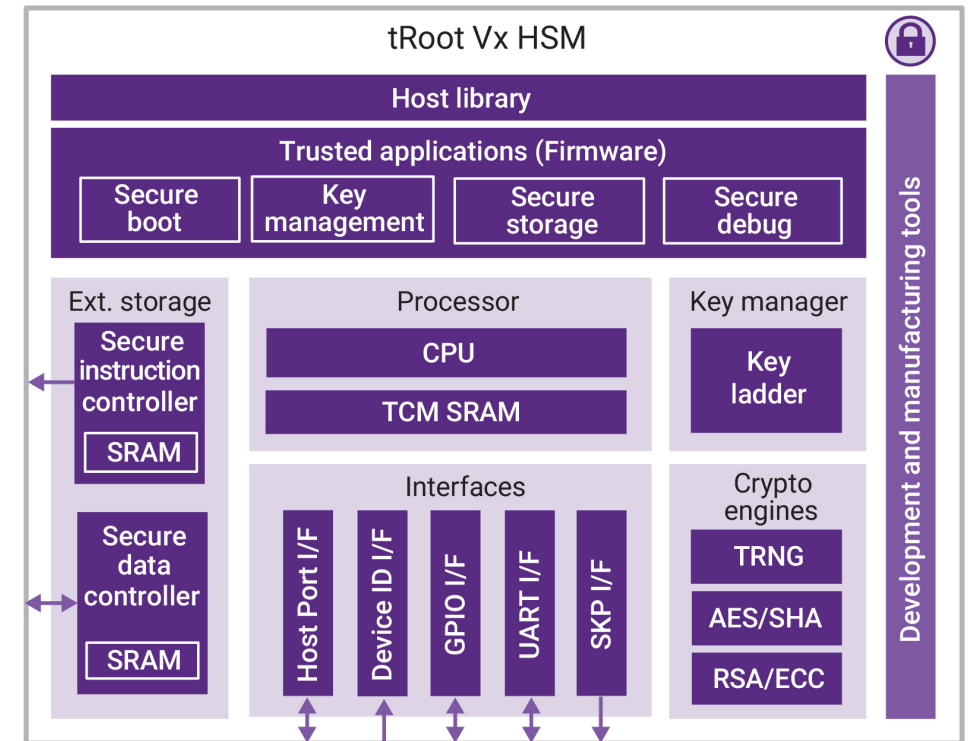
Synopsys – DesignWare tRoot Vx



Co-processor IP developed by Synopsys

- Model: tRoot V500 Hardware Secure Module
- Co-processor attaching to AMBA bus
- Some features advertised by the vendor:
 - crypto accelerators
 - secure debug
 - can act as slave device or master device for secure boot of the main processor

Not just platform to run TEE, but security manager for the whole system.



Thanks to **Benoît De Dinechin** for suggesting to add SoC security solutions.

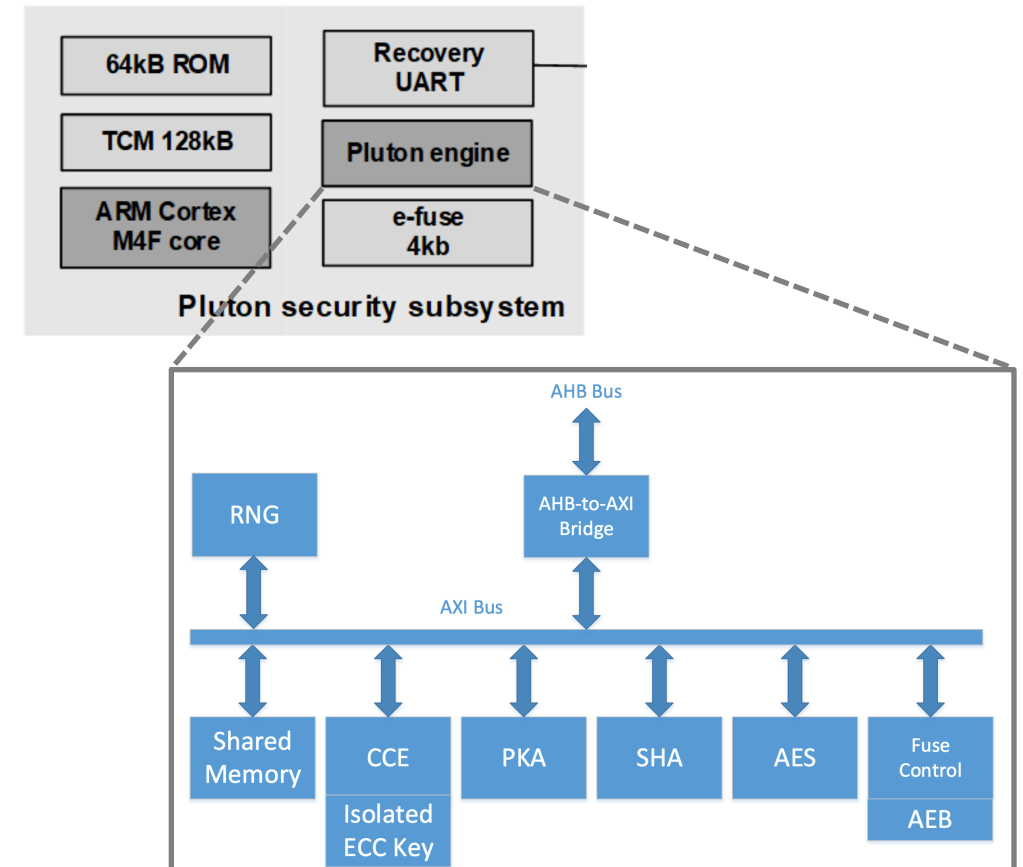
Image: <https://www.synopsys.com/dw/ipdir.php?ds=security-troot-hw-secure-module>

Microsoft – Pluton Security Subsystem



Security subsystem for Azure Sphere Microcontrollers (MCUs)

- Model: Pluton Engine
- Co-processor attaching to AHB bus
- Some features advertised by the vendor:
 - crypto accelerators



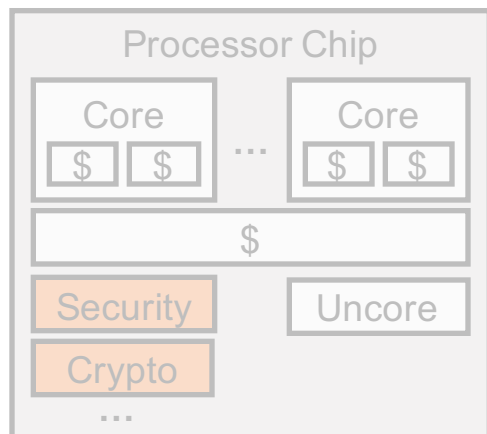
Thanks to **Hanjun Kim**
for links to HotChip slides about Pluton

Images: https://www.hotchips.org/hc30/1conf/1.13_Microsoft_Hardware_Security_Platform_Behind_Azure_Sphere.pdf

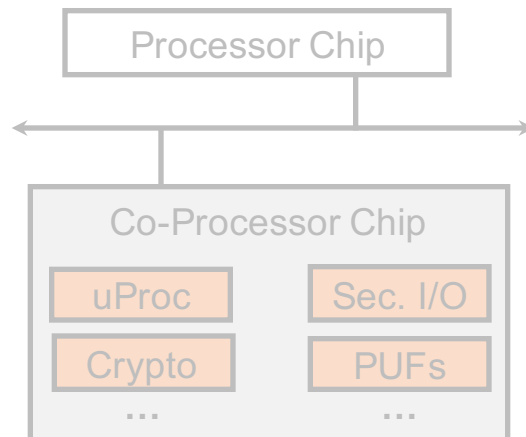
Other Security Related Architectures



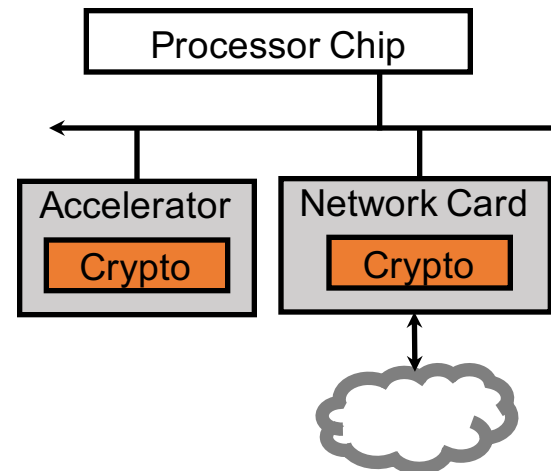
Secure Processor Architecture



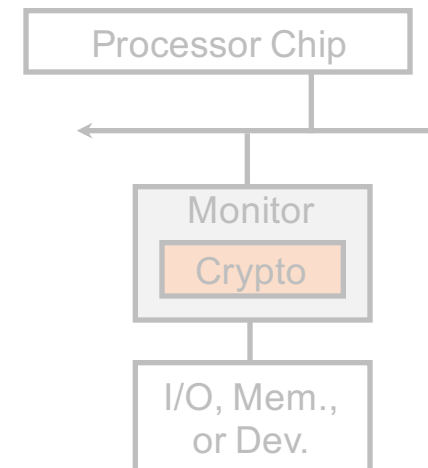
Secure Co-Processor or HSM Architecture



Cryptographic Accelerators



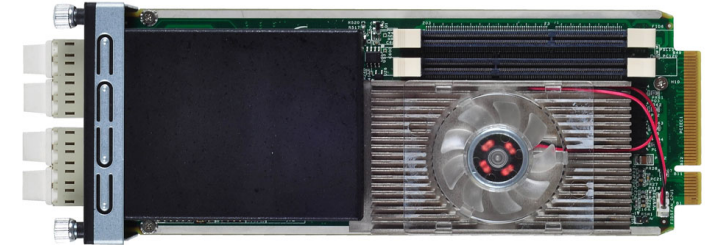
Security Monitor



Cryptographic Accelerators



- Devices for accelerating encryption or decryption
 - Network packets
 - Disk or other storage
- Can be integrated into the I/O device
 - Network card with crypto engine

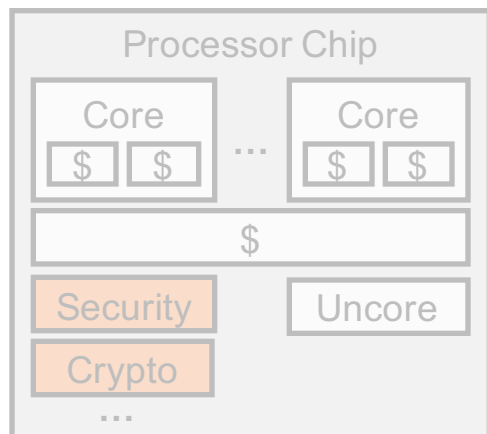


Images:
<http://www.lannerinc.com/products/x86-network-appliances/network-processing-cards/ncs-mtx401>

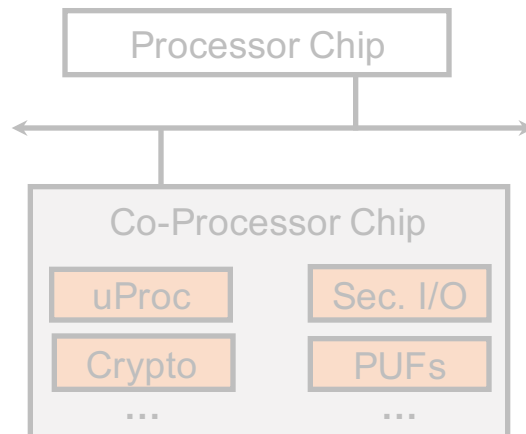
Other Security Related Architectures



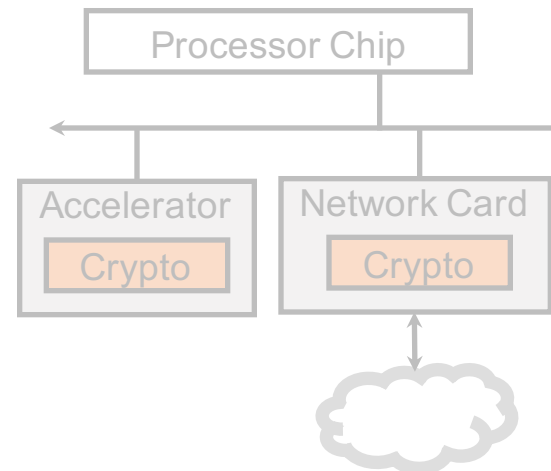
Secure Processor Architecture



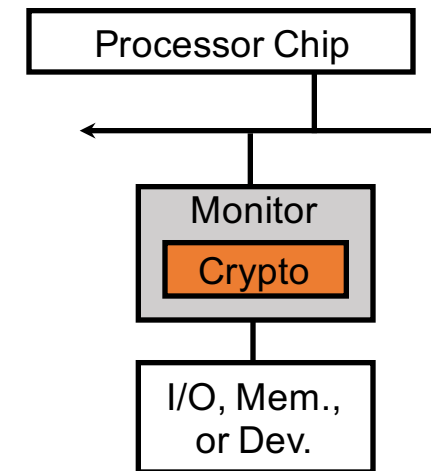
Secure Co-Processor or HSM Architecture



Cryptographic Accelerators



Security Monitor

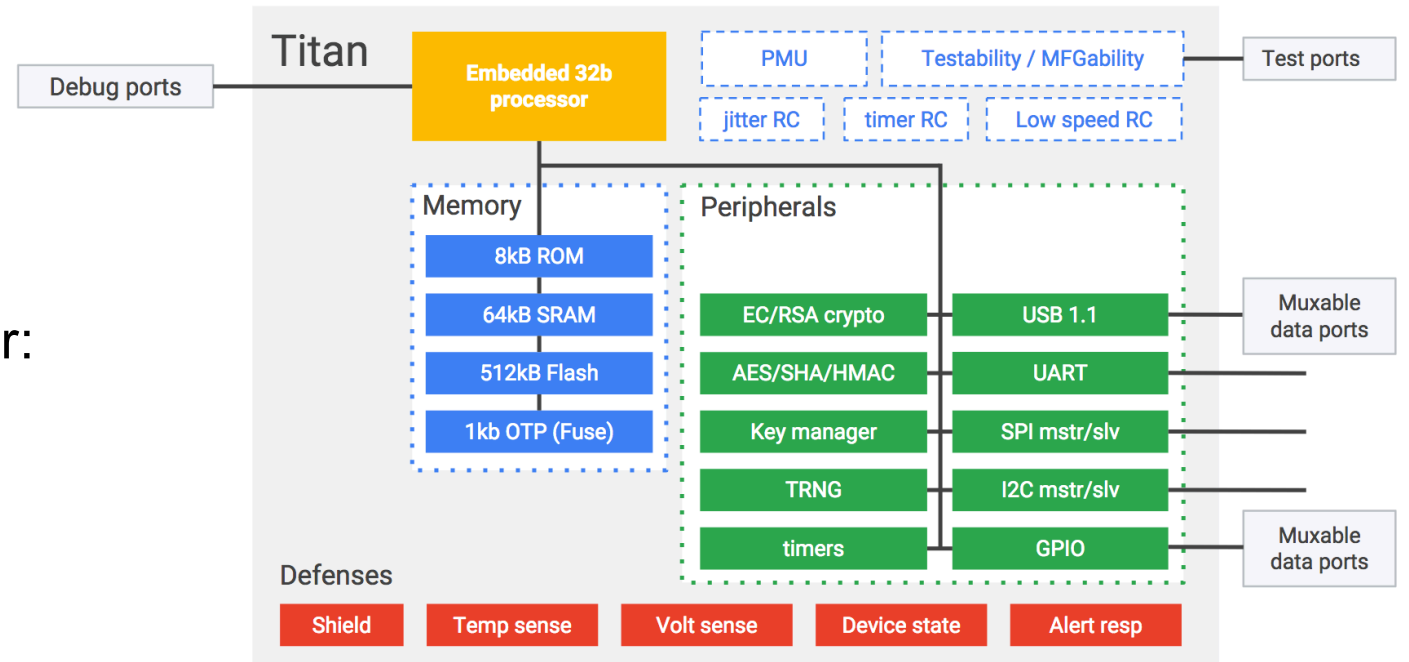


Google – Titan



Discrete chip for snooping on SPI bus and protecting Flash memory with boot ROMs

- Model: Titan
- Interposes on SPI communication to monitor status of flash memory with boot ROMs
- Some features advertised by the vendor:
 - crypto accelerators
 - attack detection (glitch, laser, thermal, voltage, or physical probing)
 - boot-time and live-status checks



Very important but different from code attestation; focuses on TRNG integrity, clock signal integrity, etc.

Thanks to **Hanjun Kim**
for links to HotChip slides about Titan

Images:
https://www.hotchips.org/hc30/1conf/1.14_Google_Titan_GoogleFinalTitanHotChips2018.pdf



Secure Processor Architectures

Trusted Execution Environments

Hardware Roots of Trust

Memory Protection

Multiprocessor and Many-core Protections

Side-Channels Threats and Protections

Principles of Secure Processor Architecture Design

Trusted Execution Environments and TCB



The goal of **Trusted Execution Environments (TEEs)** is to provide protections for a piece of code and data from a range of software and hardware attacks.

- Multiple mutually-untrusting pieces of protected code can run on a system at the same time

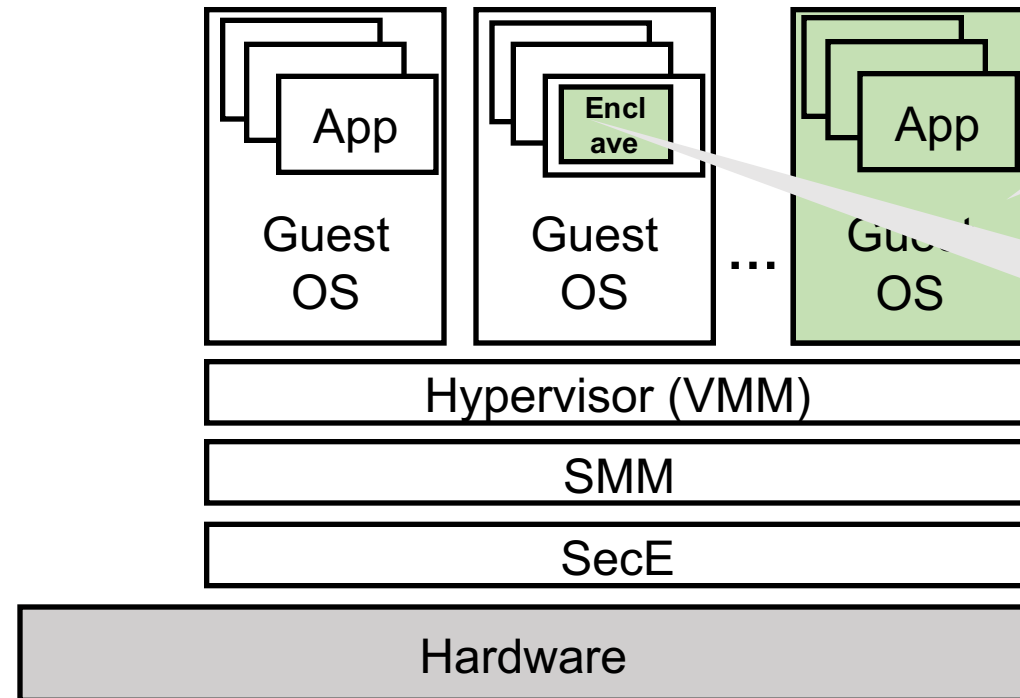
The **Trusted Computing Base (TCB)** is the set of hardware and software that is responsible for realizing the TEE:

- TEE is created by a set of all the components in the TCB
- TCB is trusted to correctly implement the protections
- Vulnerability or successful attack on TCB nullifies TEE protections

TEEs and Software They Protect



Different architectures mainly focus on **protecting Trusted Software Modules** (a.k.a. enclaves) or **whole Virtual Machines** or containers.



Some TEEs have support for protecting whole virtual machines.

Other TEEs support Trusted Software Modules, a.k.a. enclaves

Protections Offered by Secure Processor Architectures



Security properties for the TEEs that secure processor architectures aim to provide:

- Confidentiality Confidentiality is the prevention of the disclosure of secret or sensitive information to unauthorized users or entities.
- Integrity Integrity is the prevention of unauthorized modification of protected information without detection.
- Availability (next slide)

The C. I. A. properties are with respect to components or participants of the system, commonly named Alice, Bob, Charlie, Eve, Malory, etc., in different protocols

Confidentiality and integrity protections are from attacks by other components (and hardware) not in the TCB. **There is typically no protection from malicious TCB.**

Protections offered by Secure Processor Architectures



Protections not typically offered:

- Availability

Availability is the provision of services and systems to legitimate users when requested or needed.

Single processor is not able to provide availability protection (e.g. anybody can unplug computer from power source).

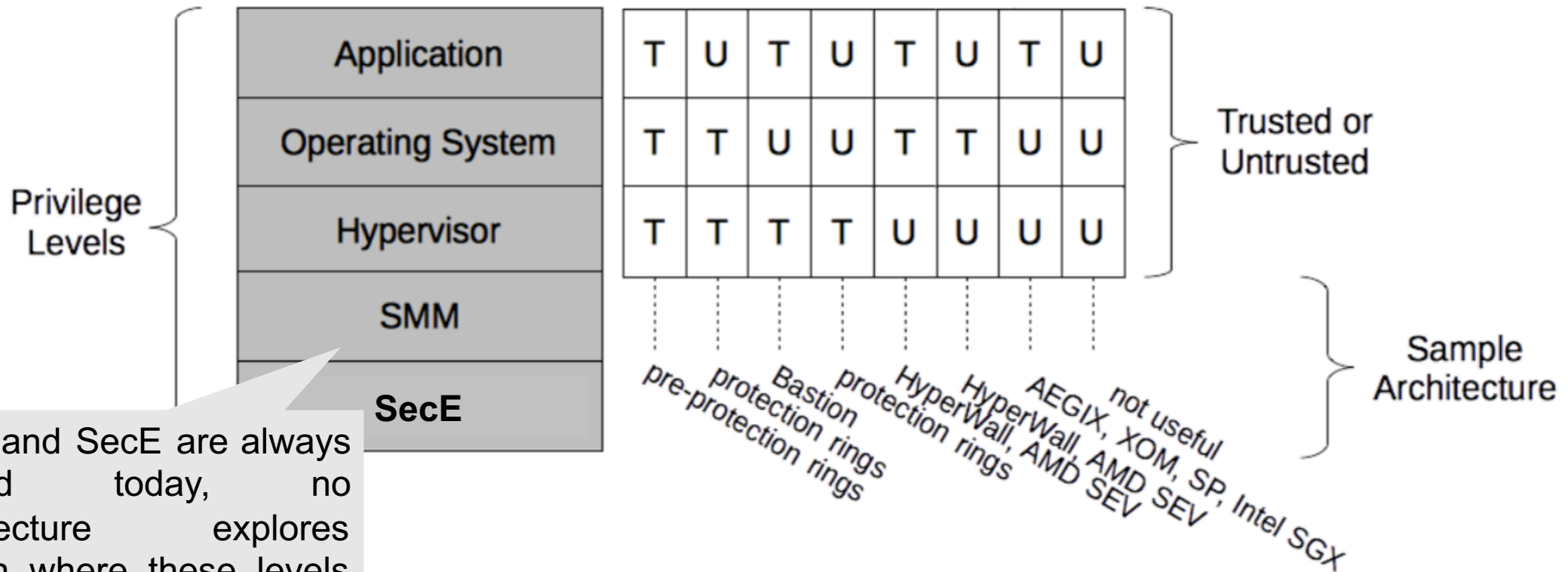
Security vs. Reliability:

Reliability protections assume random faults or errors, security protections assumes that reliability, i.e. protection from random faults or errors, is already provided by the system, and focuses instead on the deliberate attacks by a smart adversary.

Sample Protections Categorized by Architecture



Secure processor architectures break the linear relationship (where lower level protection ring is more trusted):

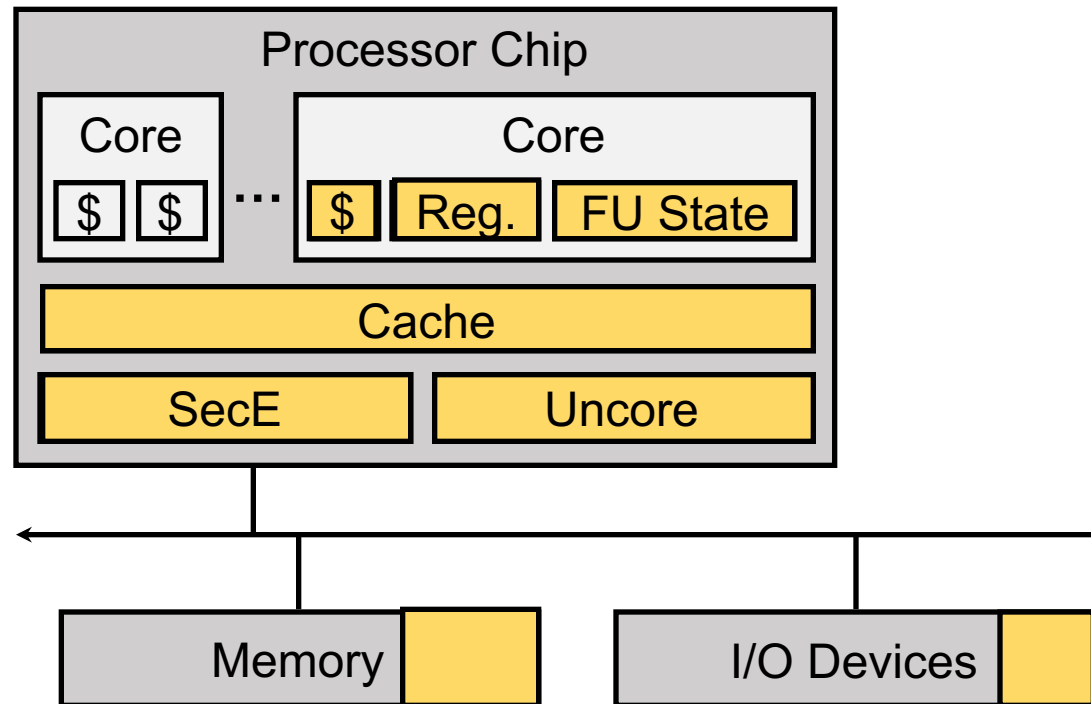


SMM and SecE are always trusted today, no architecture explores design where these levels are untrusted.

Protecting State of the Protected Software



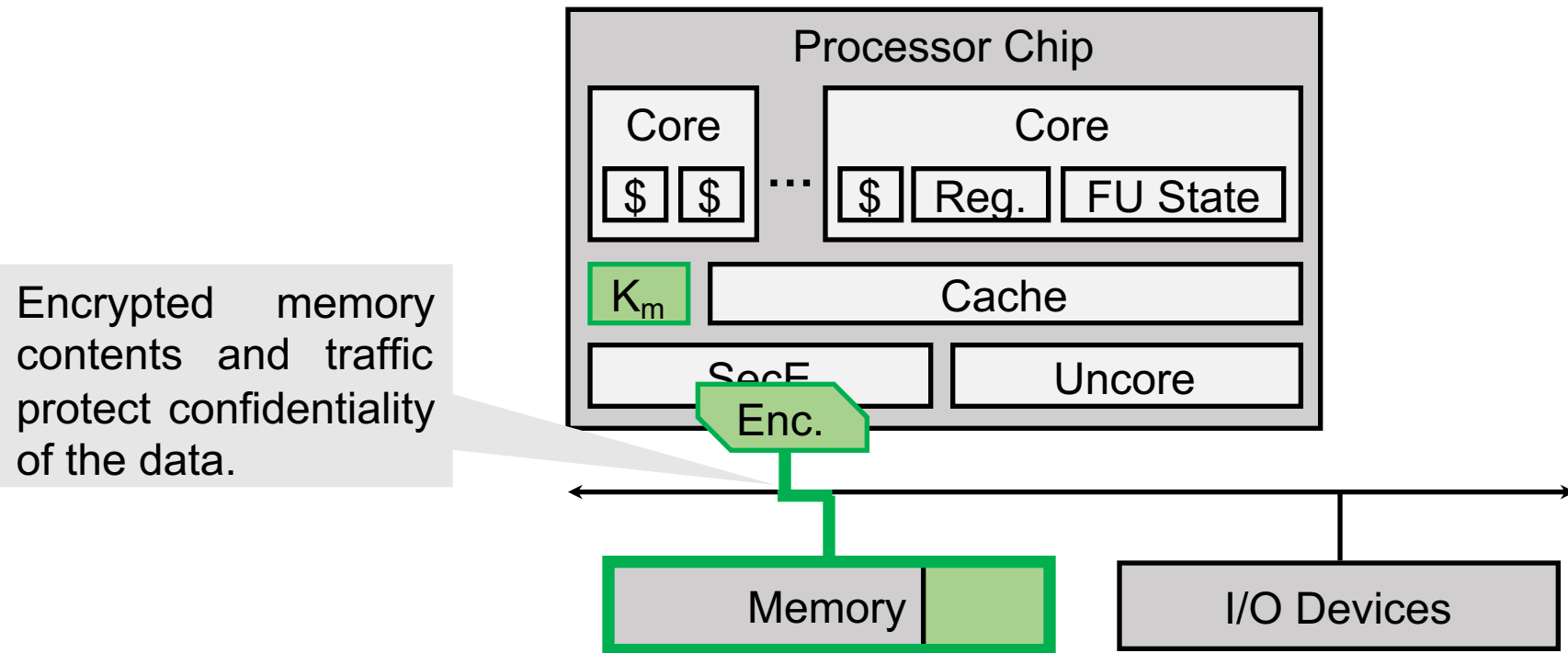
Protected software's **state** is distributed throughout the processor. All of it needs to be protected from the untrusted components and other (untrusted) protected software.



Enforcing Confidentiality through Encryption



Symmetric key cryptography should be used to protect data going off chip to prevent hardware attacks.

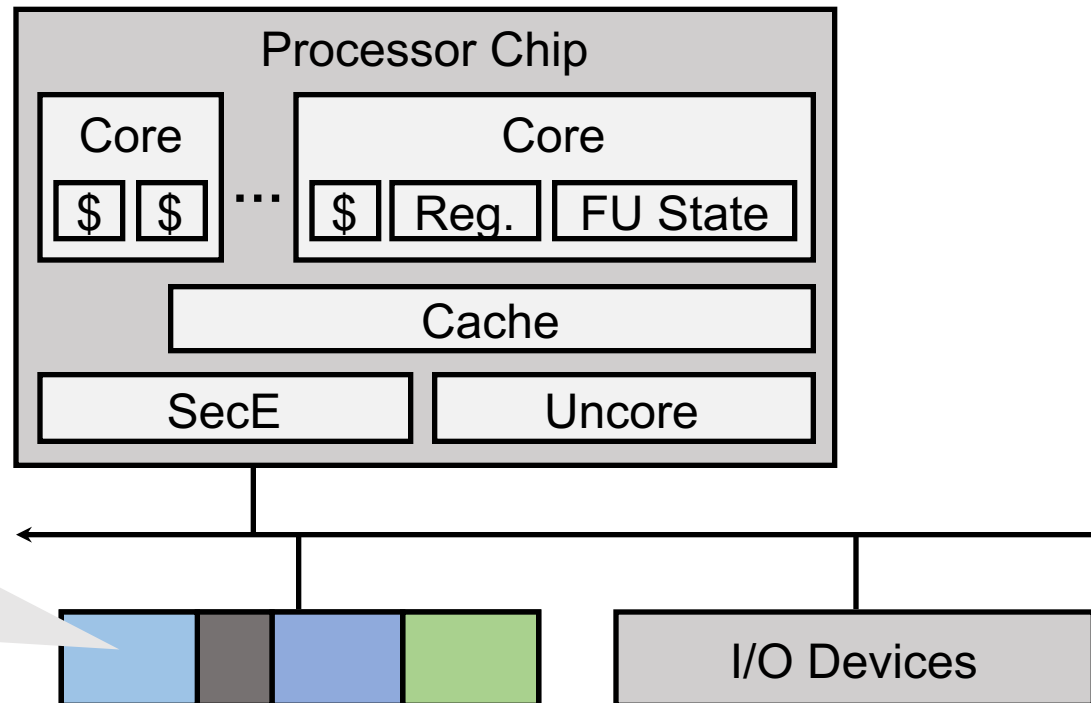


Enforcing Confidentiality through Isolation



Software entities can be separated through isolation (controlling address translation and mapping).

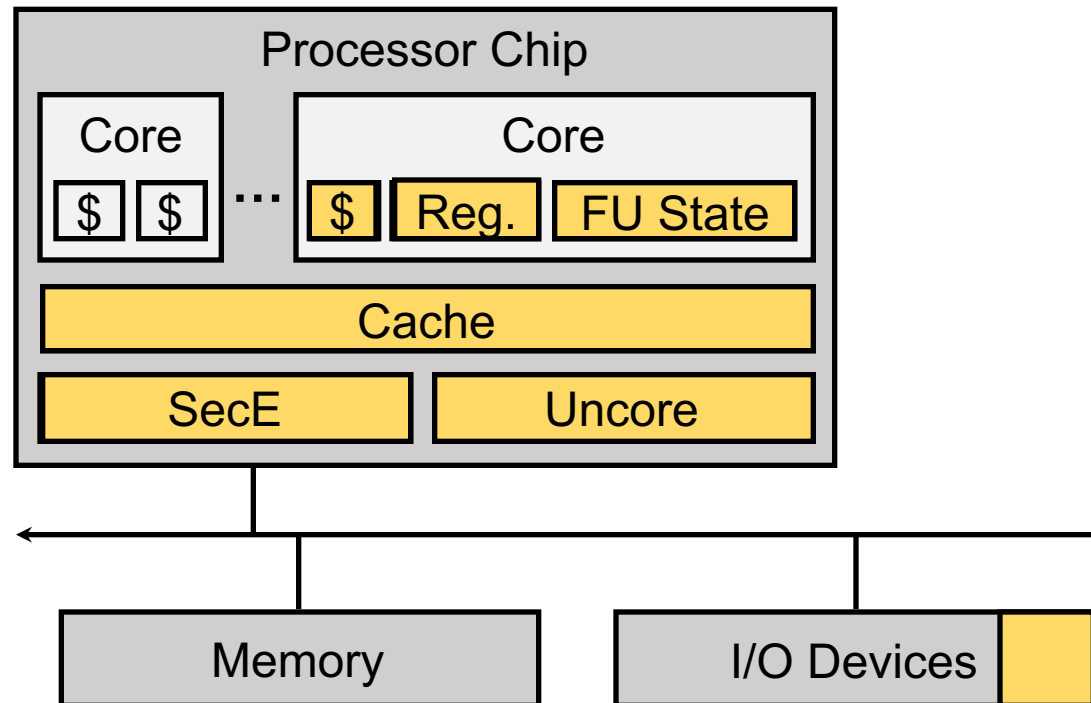
Isolating memory regions separates one software instance from each other and from untrusted software.



Enforcing Confidentiality through State Flushing



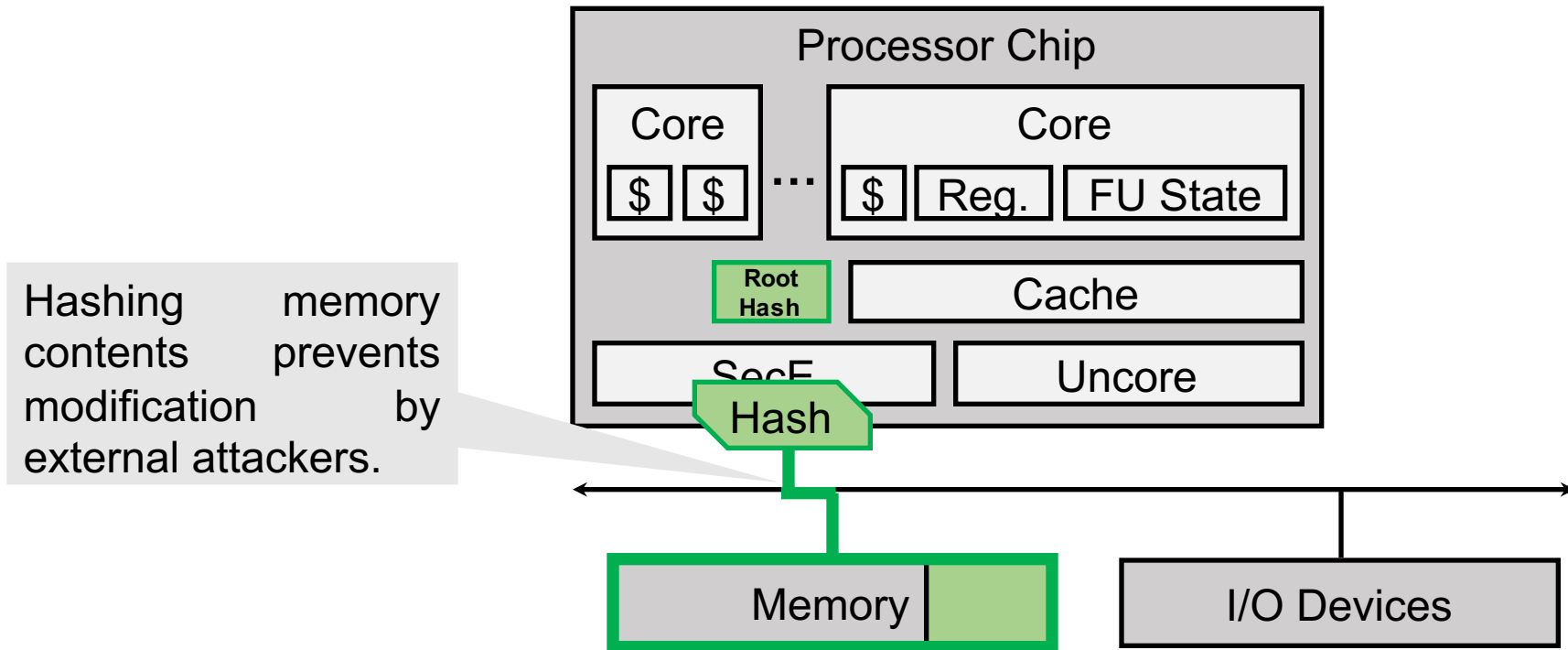
State in the processor and elsewhere in the system can be flushed to ensure confidentiality from other entities that will later run on the system.



Enforcing Integrity through Cryptographic Hashing



Symmetric key cryptography should be used to protect data going off chip to prevent hardware attacks.

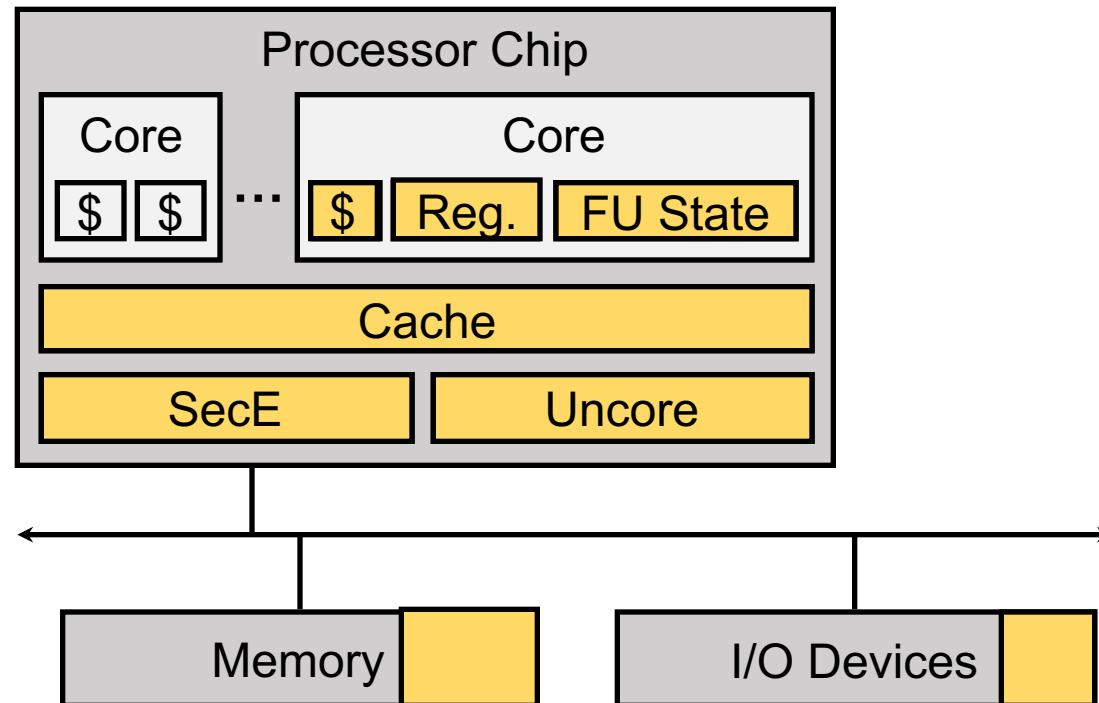


No Side-Effects Assumption



Secure processor architectures assume **no side-effects are visible to the untrusted components** whenever protected software is executing.

1. System is in some state before protected software runs
2. Protected software runs modifying system state
3. When protected software is interrupted or terminates the state modifications are erased



Benign Protected Software Assumption



The software (code and data) executing within TEE protections is assumed to be benign and not malicious:

- Goal of Secure Processor Architectures is to create minimal TCB that realizes a TEE within which the protected software resides and executes
- Secure Processor Architectures can not protect software if it is buggy or has vulnerabilities

Code bloat endangers invalidating assumptions about benign protected software.

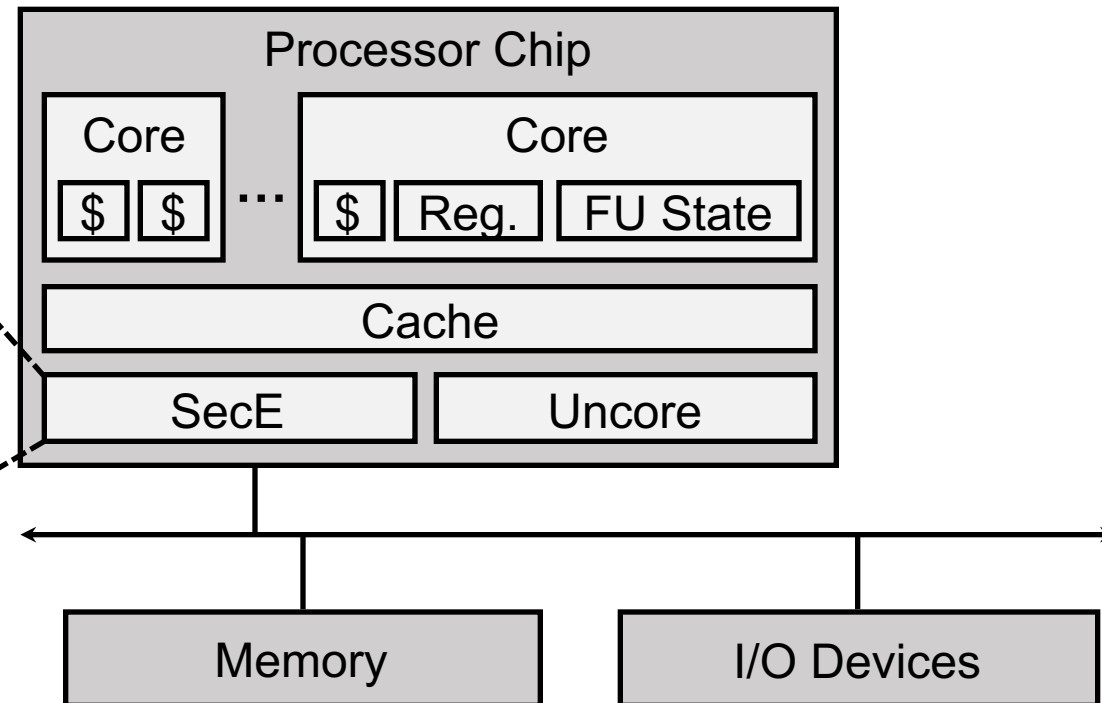
Attacks from within protected software should be defended.

Hardware TCB as Circuits or Processors



Key parts of the hardware TCB can be implemented as dedicated circuits or as firmware or other code running on dedicated processor

- **Custom logic or hardware state machine:**
 - Most academic proposals
- **Code running on dedicated processor:**
 - Intel ME = ARC processor or Intel Quark processor
 - AMD PSP = ARM processor

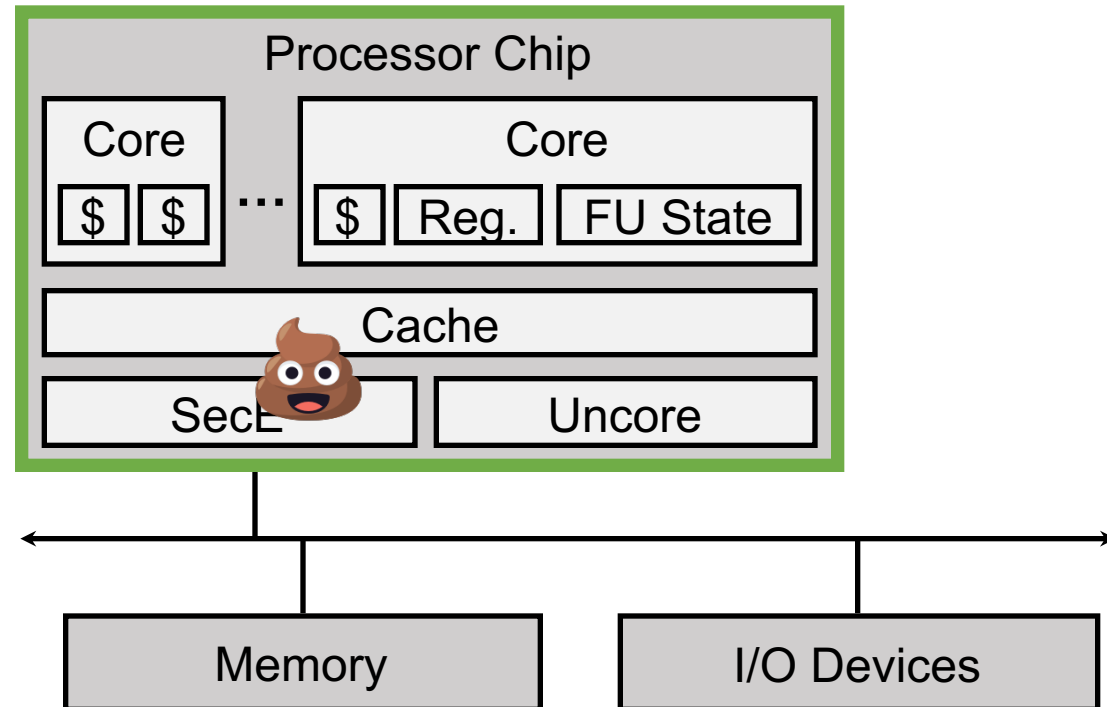


Ensuring TCB is Trustworthy



Vulnerabilities in TCB “hardware” can lead to attacks that nullify the security protections offered by the system.

- Problems in hardware state machines controlling the system
- Problem in software or firmware running on the embedded processors



Emoji Image:
<https://www.emojione.com/emoji/1f4a9>

Trustworthy TCB Execution Assumption



Trustworthiness of the TCB depends on the ability to monitor the TCB code (hardware and software) execution as the system runs.

Monitoring of TCB requires mechanisms to:

- Fingerprint and authenticate TCB code
- Monitor TCB execution
- Protect TCB code (on embedded security processor)
 - Virtual Memory, ASLR, ...

Performance Overhead of Securing TCB



Impact of threat model on performance:

- Protecting against more threats typically adds more overhead
- Memory encryption and integrity checking are the most expensive part, but really depends on how defense is implemented
- Secure caches: 1~10% overhead
- Spectre protections: initially stated $>10\%$, now most $<10\%$
- Memory encryption: can be $>100\%$

More protections, must not mean less performance:

- Partitioning
- Randomization is not always bad

Alternatives: FHE, FE, ...



TEEs use trusted hardware and software to protect computation that is done in **plaintext**.

Cryptography-based approaches could be used, but they come at tremendous performance cost and are not practical today.

	Obf.	FHE	FE	MPC	RE or Garbling	GES
Input	Plaintext	Ciphertext	Ciphertext	Ciphertext	Ciphertext	Ciphertext
Output	Plaintext	Ciphertext	Plaintext	Plaintext	Plaintext	Ciphertext or 0
Is the function public?	No	Yes	Usually Yes	Yes	No	Yes

- **FHE** – Fully Homomorphic Encryption
- **FE** – Function Encryption
- **MPC** – Multi-Party Computation
- **RE** – Randomized Encodings
- **GES** – Graded Encoding Scheme



Secure Processor Architectures
Trusted Execution Environments

14:00 – 14:10 Break

Hardware Roots of Trust

Memory Protection

Multiprocessor and Many-core Protections

Side-Channels Threats and Protections

Principles of Secure Processor Architecture Design

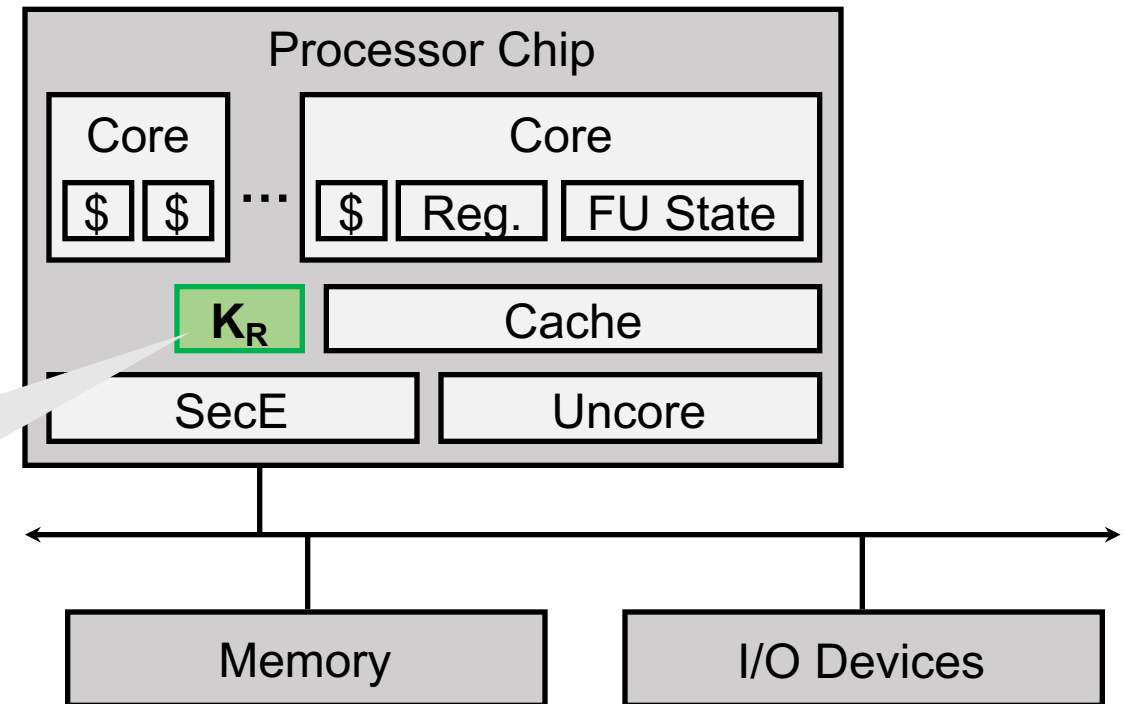
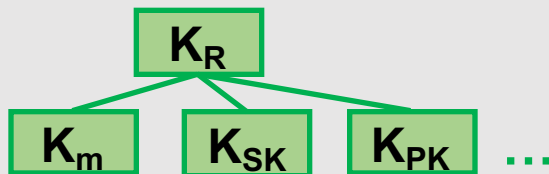
Root of Trust and the Processor Key



Security of the system is derived from a **root of trust**.

- A secret (cryptographic key) only accessible to TCB components
- Derive encryption and signing keys from the root of trust

Hierarchy of keys can be derived from the root of trust

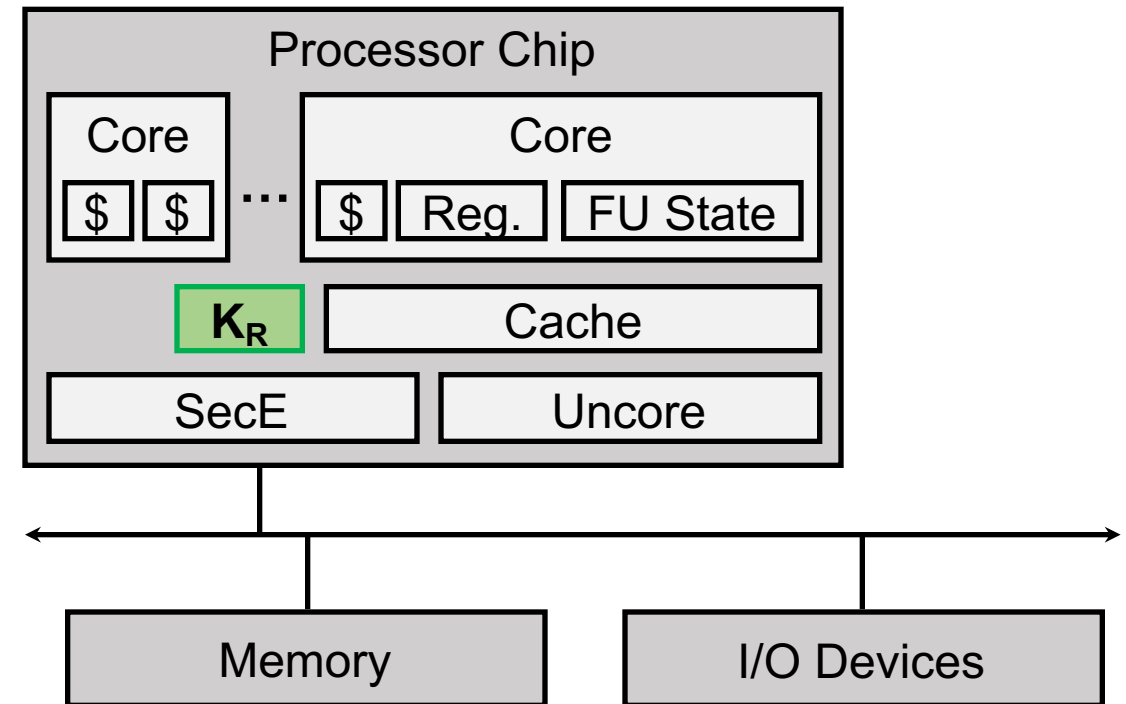


Root of Trust and Processor Key



Each processor requires a unique secret.

- **Burn in at the factory** by the manufacturer (but implies trust issues with manufacturer and the supply chain)
 - E.g. One-Time Programmable (OTP) fuses
- Use **Physically Uncloneable Functions** (but requires reliability)
 - Extra hardware to derive keys from PUF
 - Mechanisms to generate and distribute certificates for the key



Secrecy of Root of Trust Key Assumption



The unique processor key is assumed to be never disclosed to anybody.

- Manufacturer protects the keys
- Manufacturer is trusted to never disclose the keys

If using PUFs, then the trusted party doing the enrollment and key generation is trusted

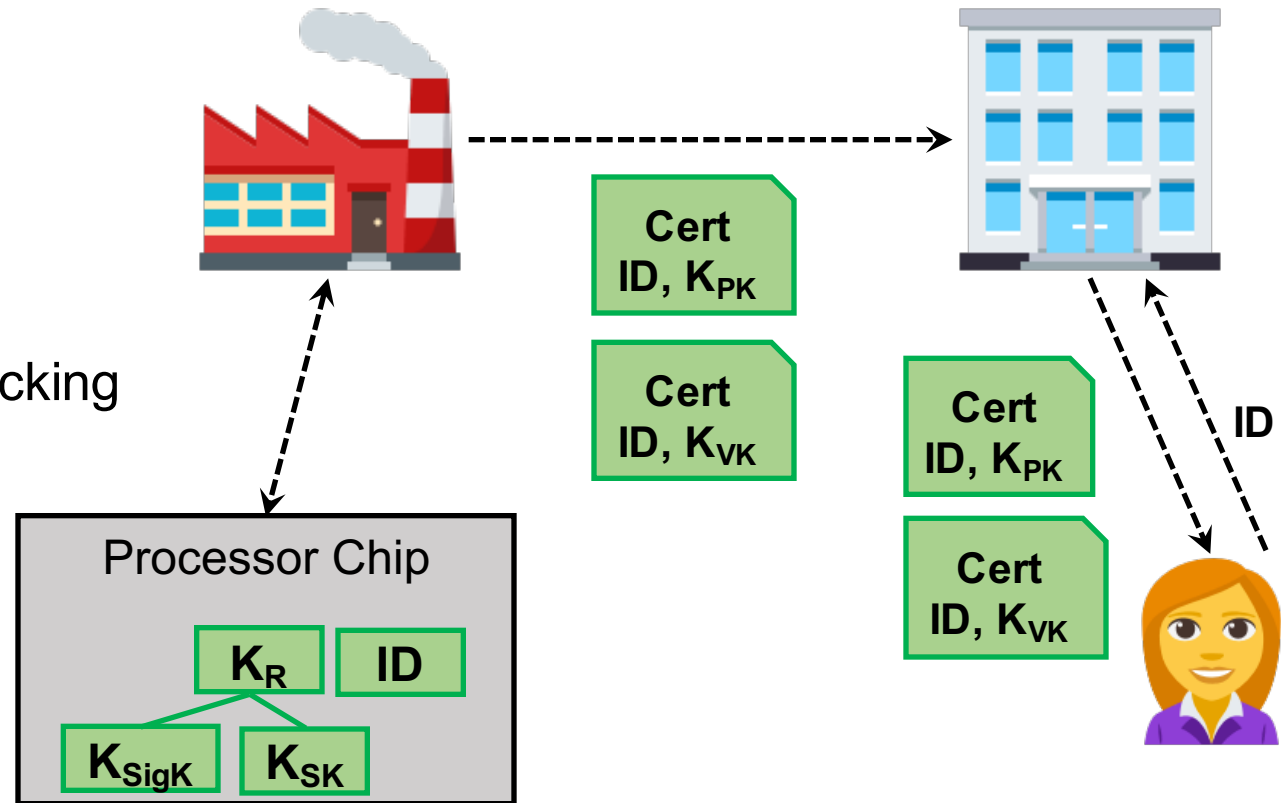
- Trust enrolling party
- Or may need on-chip key generation facility

Derived Keys and Key Distribution



Derived from the root of trust are signing and verification keys.

- Public key, K_{PK} , for encrypting data to be sent to the processor
 - Data handled by the TCB
- Signature verification key, K_{VK} , for checking data signed by the processor
 - TCB can sign user keys

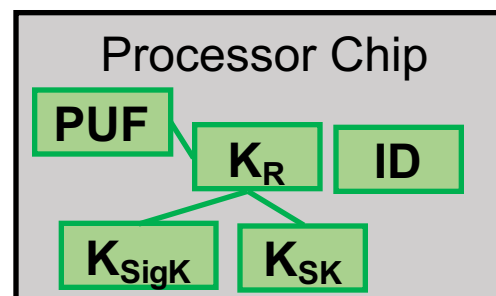


Key Distribution for PUF-based Designs



Designs that leverage PUF may require users or companies to run their own key distribution solutions.

- Deploy own infrastructure
- Use a trusted 3rd party



Protected Root of Trust Assumption



The root of trust is assumed to be protected.

If keys are **burned-in by the manufacturer**

- Secret keys are only known to the manufacturer
- Manufacturer keeps secure database of the keys

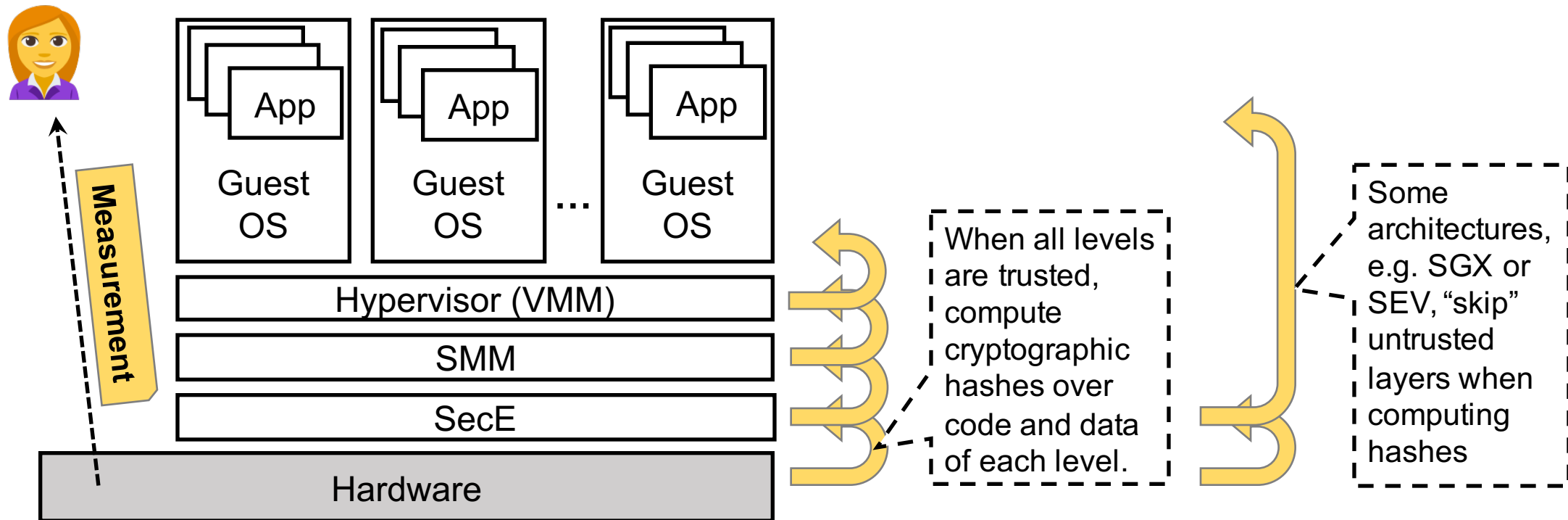
If keys are derive from **PUFs**:

- Keys are certificates are generated on-chip
- Or, generated keys are only available to trusted enrolling party
- New keys can be regenerated or it is known if key was already generated and "locked"

Software Measurement



With an embedded signing key, the software running in the TEE can be “measured” to attest to external users what code is running on the system.



Emoji Image:
<https://www.emojione.com/emoji/1f469-1f4bc>

Trusted / Secure / Authenticated Boot



When the system boots up, the software components of the TCB are measured:

- Abort when wrong measurement is obtained
- Or, continue booting but do not decrypt secrets

Any single bit change in the TCB software will give different measurement, and prevent correct bootup:

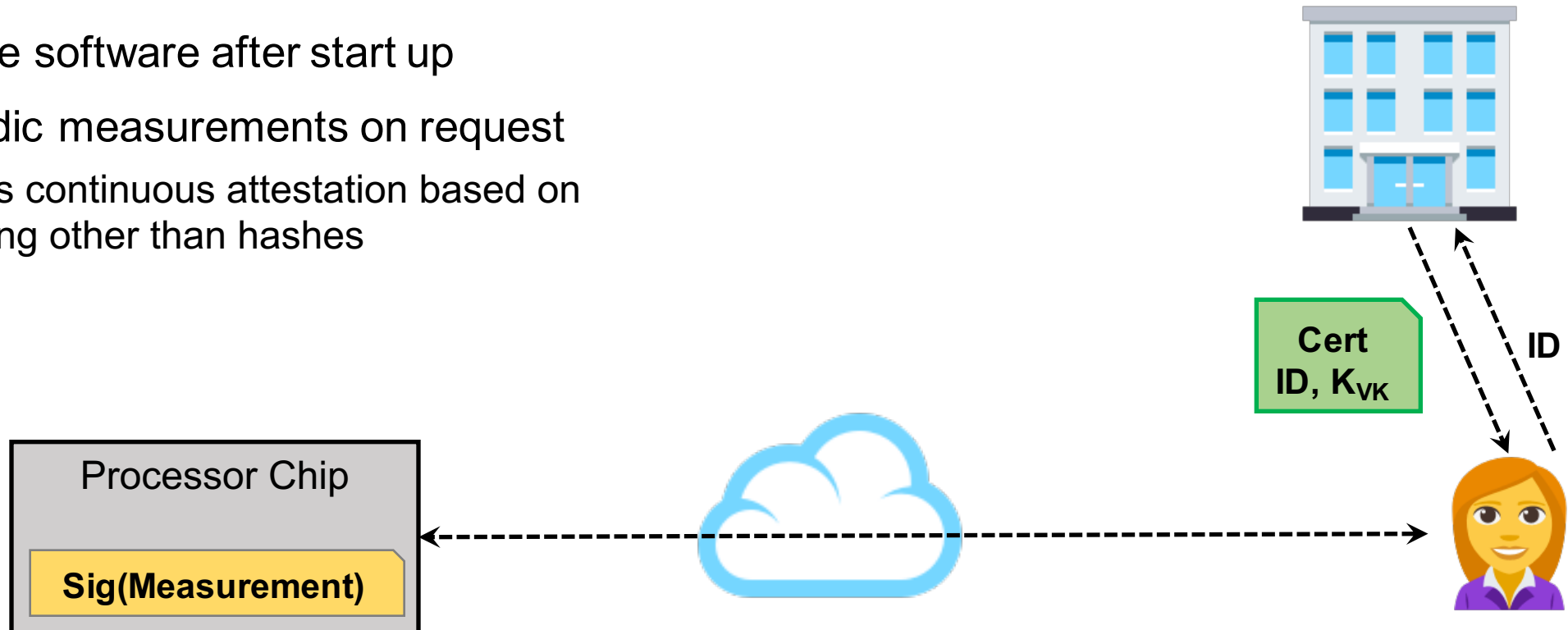
- Legitimate software updates will change measurements

Remote Attestation



TCB can sign measurements taken and send a digital signature to the remote user:

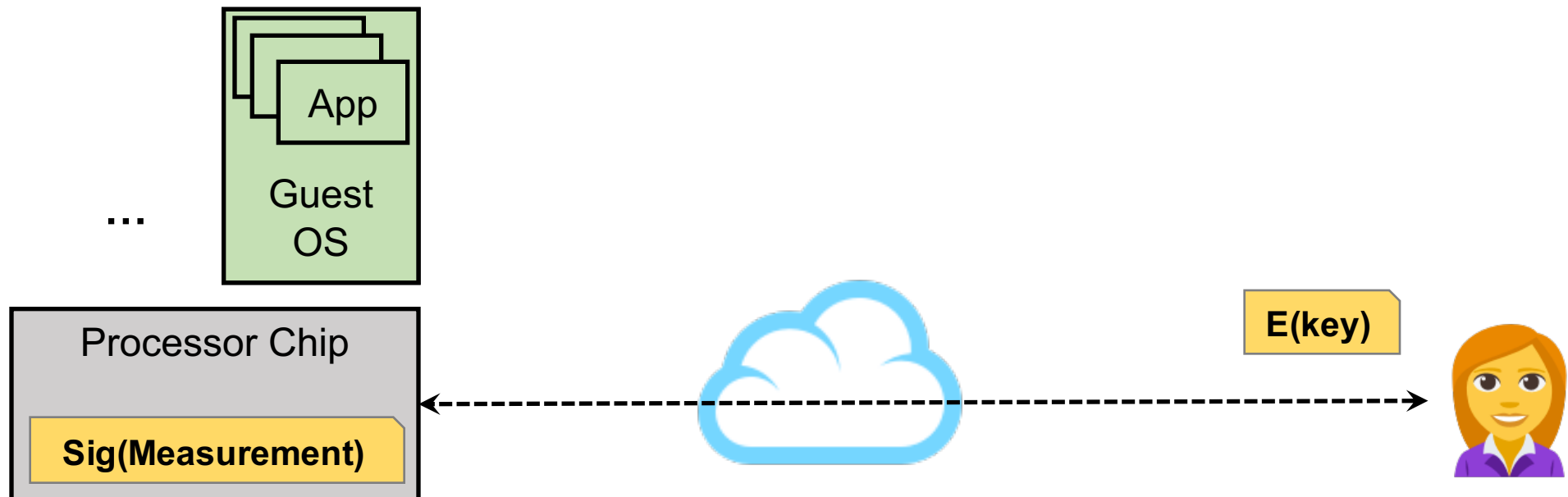
- Measure the software after start up
- Send periodic measurements on request
 - Requires continuous attestation based on something other than hashes



Data Sealing (Remote)



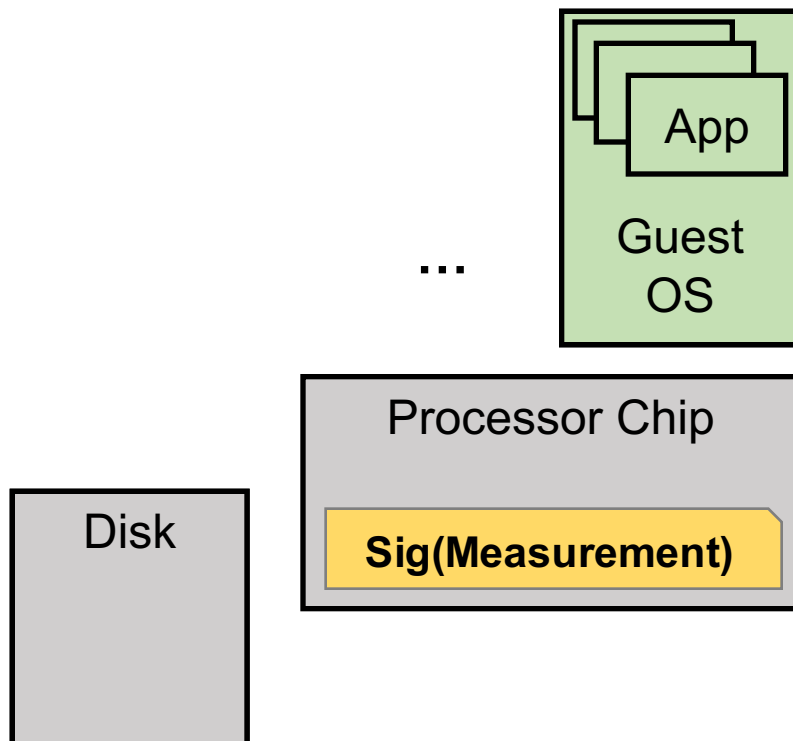
Data can be sealed (encrypted) and correct decryption key can be only made available once a measurement is verified.



Data Sealing (Local)



Locally, the measurement, taken by the TCB, can be used to unlock data on storage such as on hard disk (e.g. BitLocker).



TOC-TOU Attacks and Measurements



Time-of-Check to Time-of-Use (TOC-TOU) attacks leverage the delay between when a measurement is taken, and when the component is used.

- System can be compromised
- But measurement indicates correct data

Cannot easily use hashes to prevent TOC-TOU attacks, as one would have to have reference hashes for all different possible runtime states of the software.

Continuous Monitoring of Protected Software



Continuous monitoring is potential solution to TOC-TOU:

- Constantly measure the system, e.g. performance counters, and look for anomalies
- Requires knowing correct and expected behavior of system
- Can be used for continuous authentication

Attacker can “hide in the noise” if they change the execution of the software slightly and do not affect performance counters significantly.

Fresh Measurement Assumption



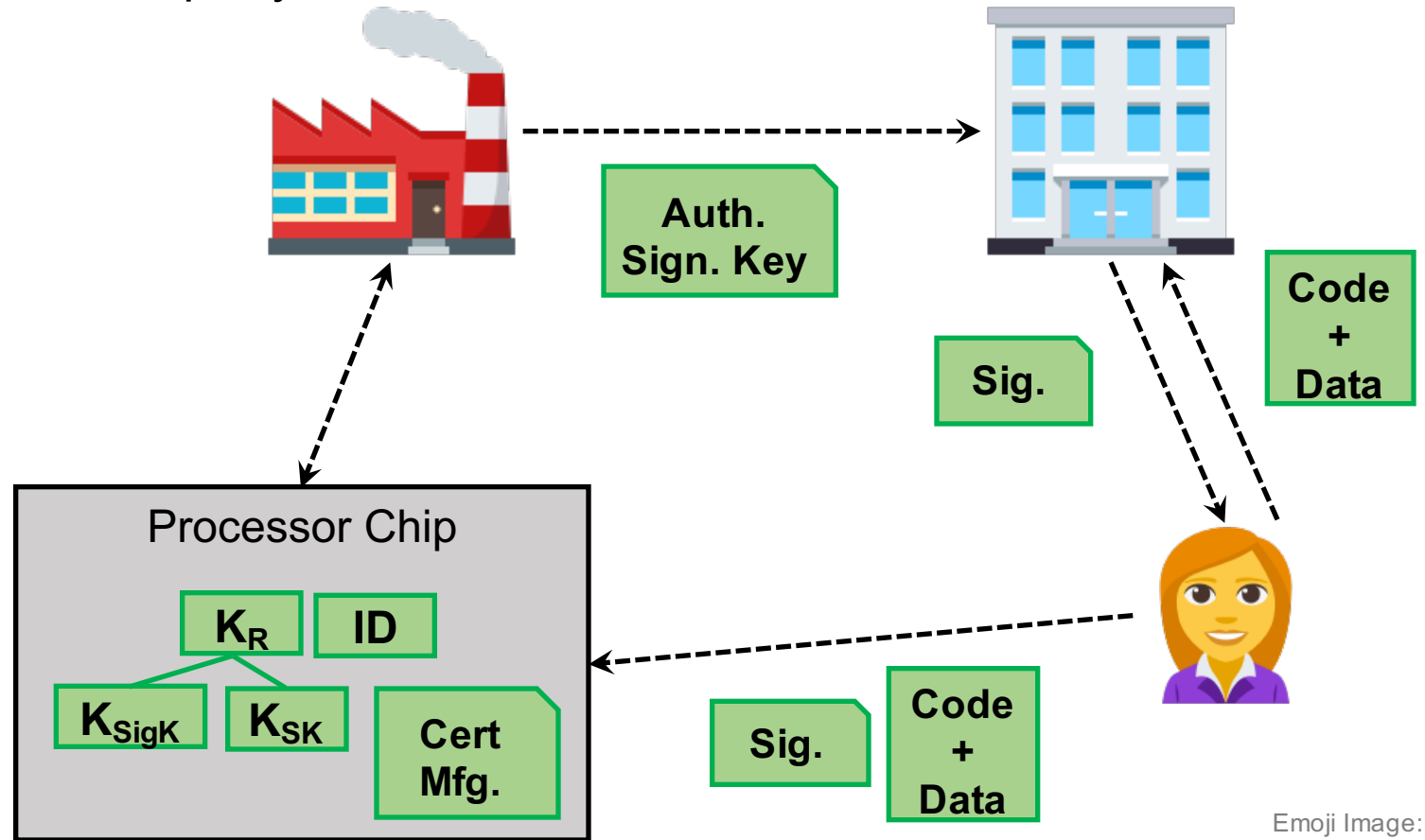
Authentication and data sealing give access to data to correctly executing software.

- Measurements used to un-seal data need to be fresh
- Revoke access if measurements change
 - But data may have already leaked out

Limiting Execution to only Authorize Code



Firmware (TCB) updates or protected software can be authenticated in the processor through use of signatures made by a trusted party.



Privacy and Lock-in Concerns



Privacy issue arise from the authentication mechanisms:

- If using private key directly each time, can know from which processor are the messages coming
- If the Certificate Authority is run by the manufacturer, they know exactly when the processor is being used

Direct Anonymous Attestation (DAA) from TPM offers some protections while allowing for remote authentication.

Lock-in issues arise from limiting what code can run on the system:

- Signature is required by 3rd party to get firmware update or software to run
- Depend on 3rd party for approval

Secure Processor Architectures
Trusted Execution Environments
Hardware Roots of Trust

Memory Protection

Multiprocessor and Many-core Protections
Side-Channels Threats and Protections
Principles of Secure Processor Architecture Design

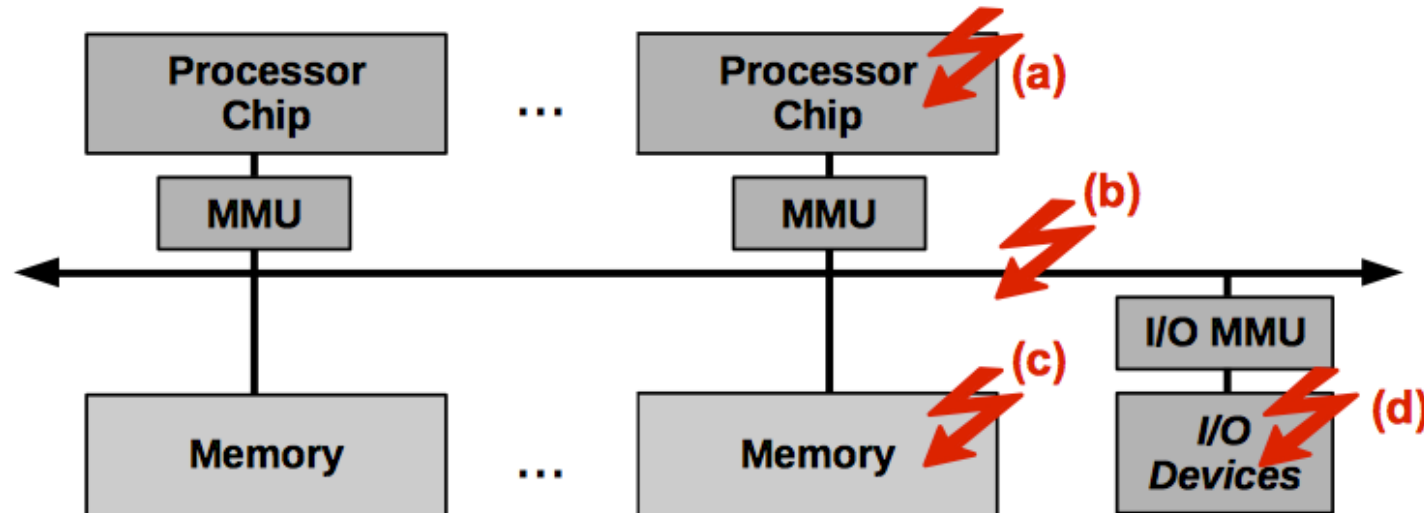


Sources of Attacks on Memory



Memory is vulnerable to different types of attacks:

- Untrusted software running on the processor
- Physical attacks on the memory bus, other devices snooping on the bus, man-in-the-middle attacks with malicious device
- Physical attacks on the memory (Coldboot, ...)
- Malicious devices using DMA or other attacks



Types of Attacks on Memory



Different types of attacks exist (very similar to attacks in network settings):

- Snooping

Passive attack, try to read data contents.
- Spoofing

Active attack, inject new memory commands to try to read or modify data.
- Splicing

Active attack, combine portions of legitimate memory commands into new memory commands (to read or modify data).
- Replay

Active attack, re-send old memory command (to read or modify data).
- Disturbance

Active attack, DoS on memory bus, repeated memory accesses to age circuits, repeated access to make Rowhammer, etc.

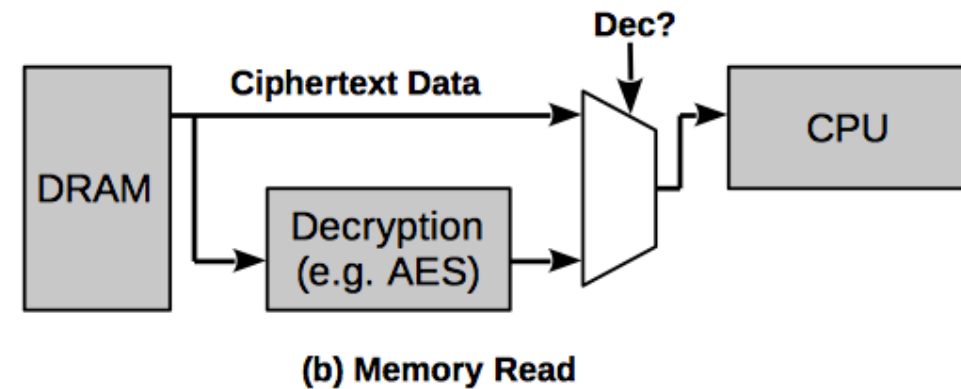
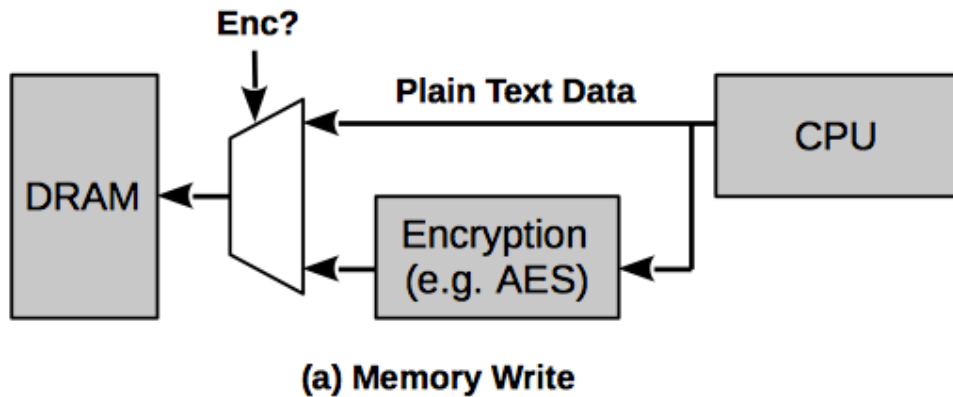
Confidentiality Protection with Encryption



Contents of the memory can be protected with encryption. Data going out of the CPU is encrypted, data coming from memory is decrypted before being used by CPU.

- a) Encryption engine (usually AES in CTR mode) encrypts data going out of processor chip
- b) Decryption engine decrypts incoming data

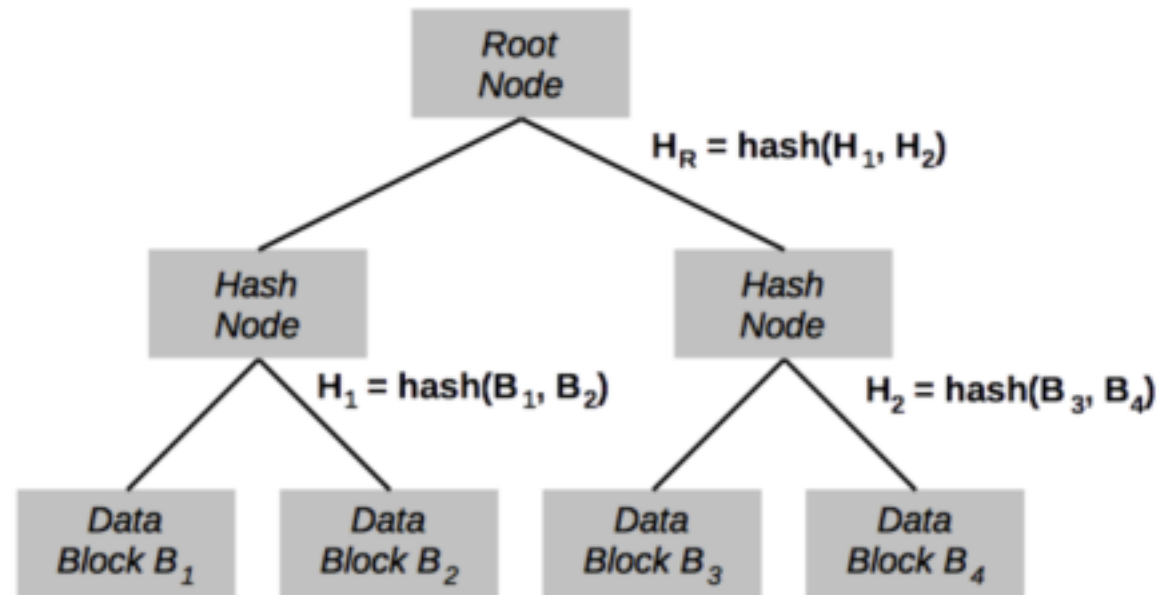
Pre-compute encryption pads, then only need to do XOR; speed depends on how well counters are fetched / predicted.



Integrity Protection with Hash Trees



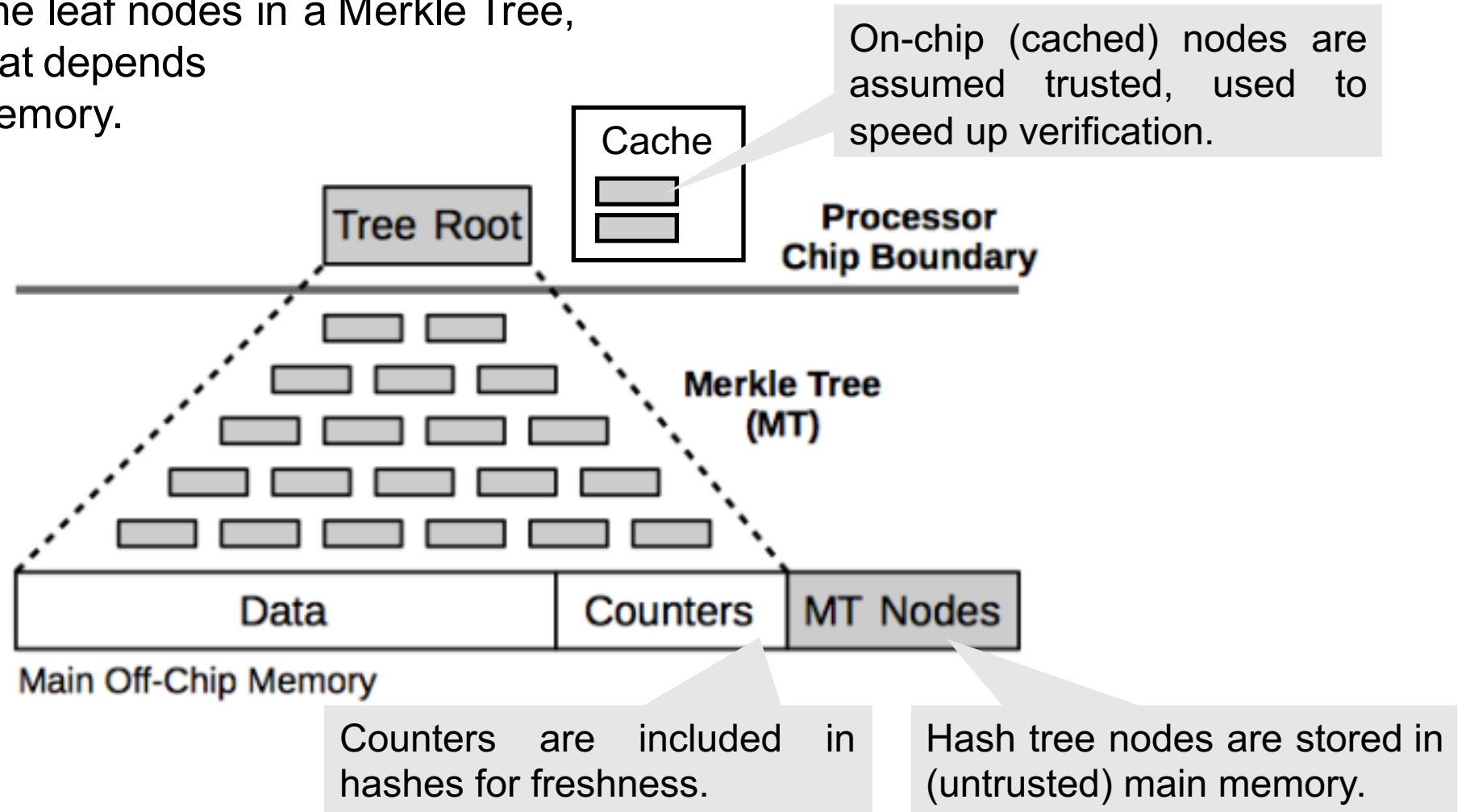
Hash tree (also called **Merkle Tree**) is a logical tree structure, typically a binary tree, where two child nodes are hashed together to create parent node; the root node is a hash that depends on value of all the leaf nodes.



Integrity Protection with Hash Trees



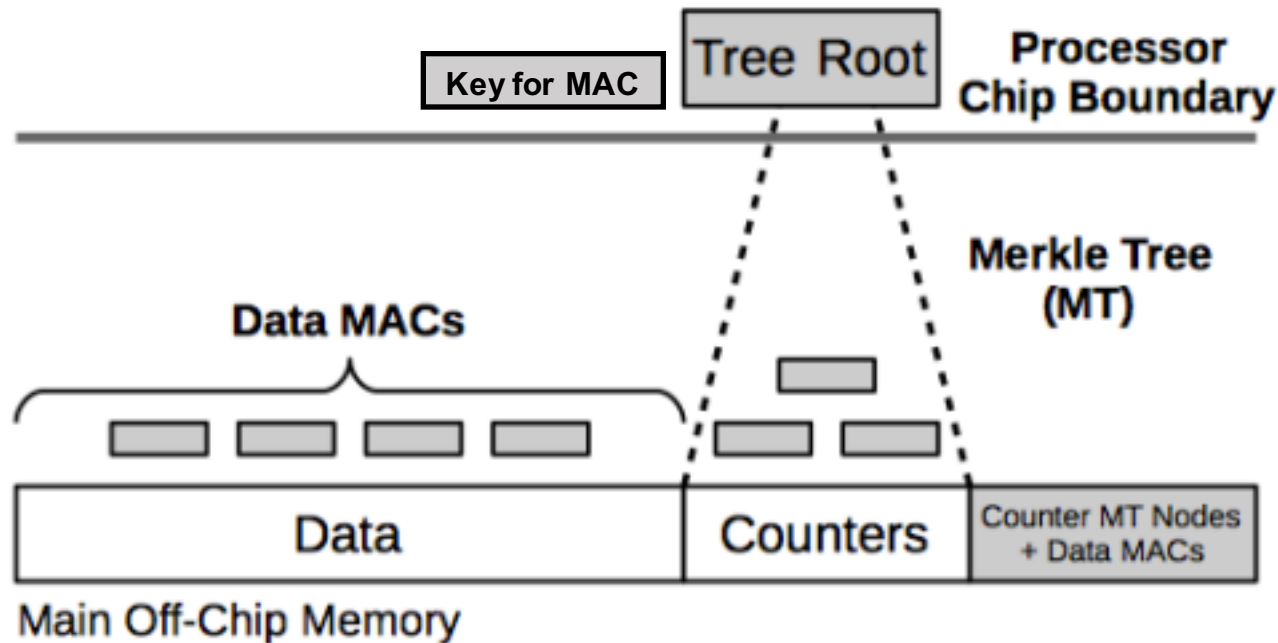
Memory blocks can be the leaf nodes in a Merkle Tree, the tree root is a hash that depends on the contents of the memory.



Integrity Protection with Bonsai Hash Trees



Message Authentication Codes (MACs) can be used instead of hashes, and a smaller “Bonsai” tree can be constructed.

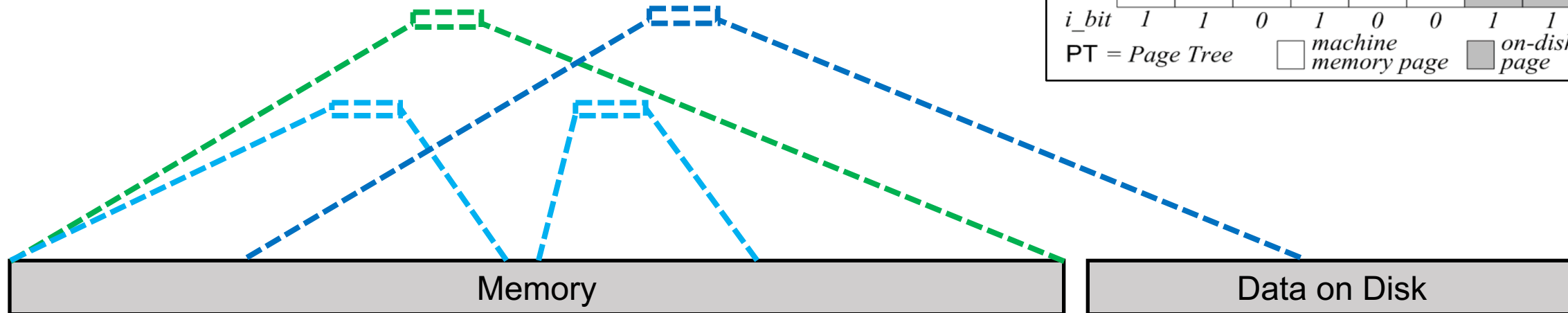
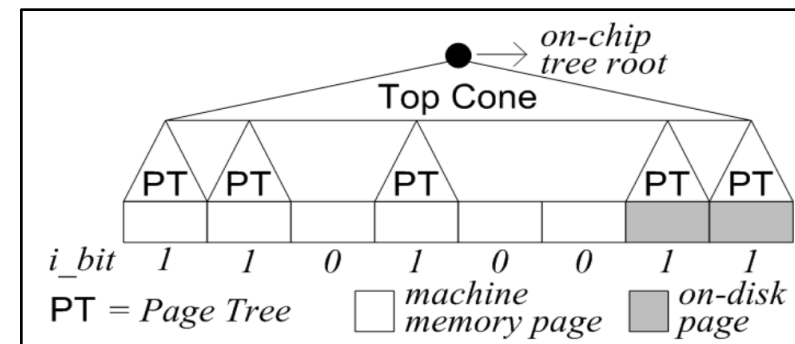


Integrity Protection of Selected Memory Regions



- For encryption, type of encryption does not typically depend on memory configuration
- For integrity, the integrity tree needs to consider:
 - Protect whole memory
 - Protect parts of memory (e.g. per application, per VM, etc.)
 - Protect external storage (e.g. data swapped to disk)

E.g., Bastion's memory integrity tree (Champagne, et al., HPCA '10)

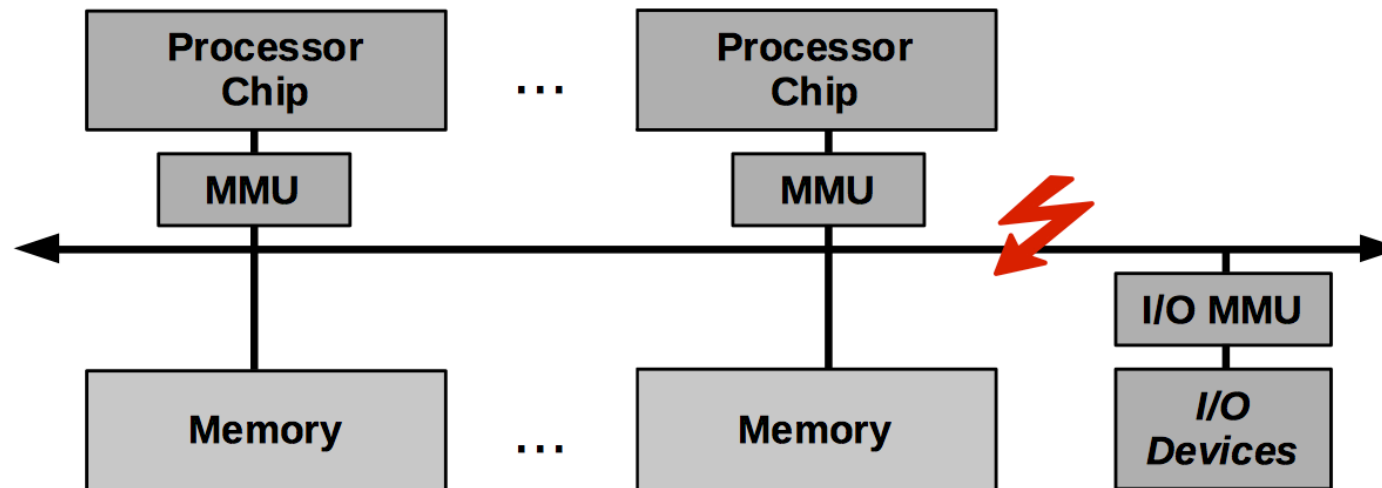


Memory Access Pattern Protection



Snooping attacks can target extracting data (protected with encryption) or **extracting access patterns** to learn what a program is doing.

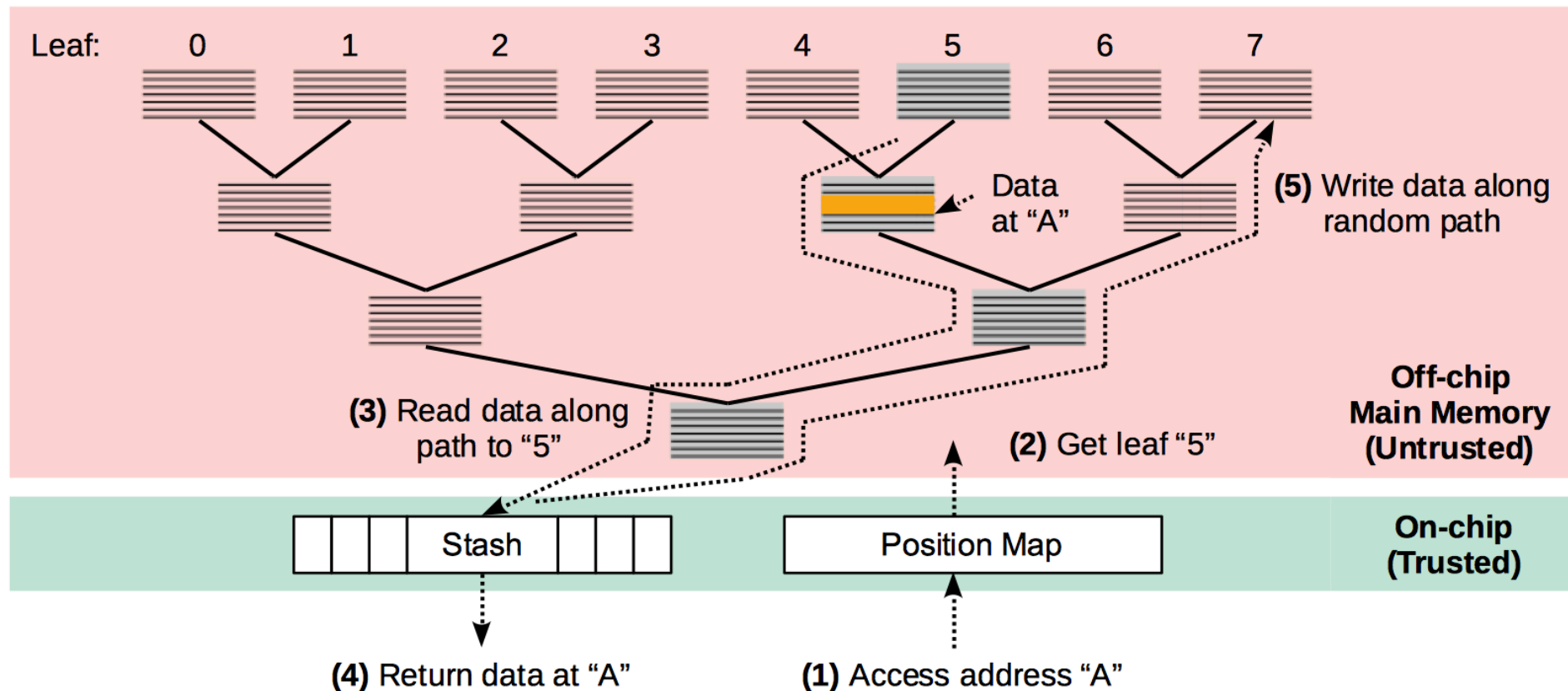
- Easier in Symmetric multiprocessing (SMP) due to shared bus
- Possible in other configuration if there are untrusted components



Memory Access Pattern Protection



Access patterns (traffic analysis) attacks can be protected with use Oblivious RAM, such as Path ORAM. This is on top of encryption and integrity checking.

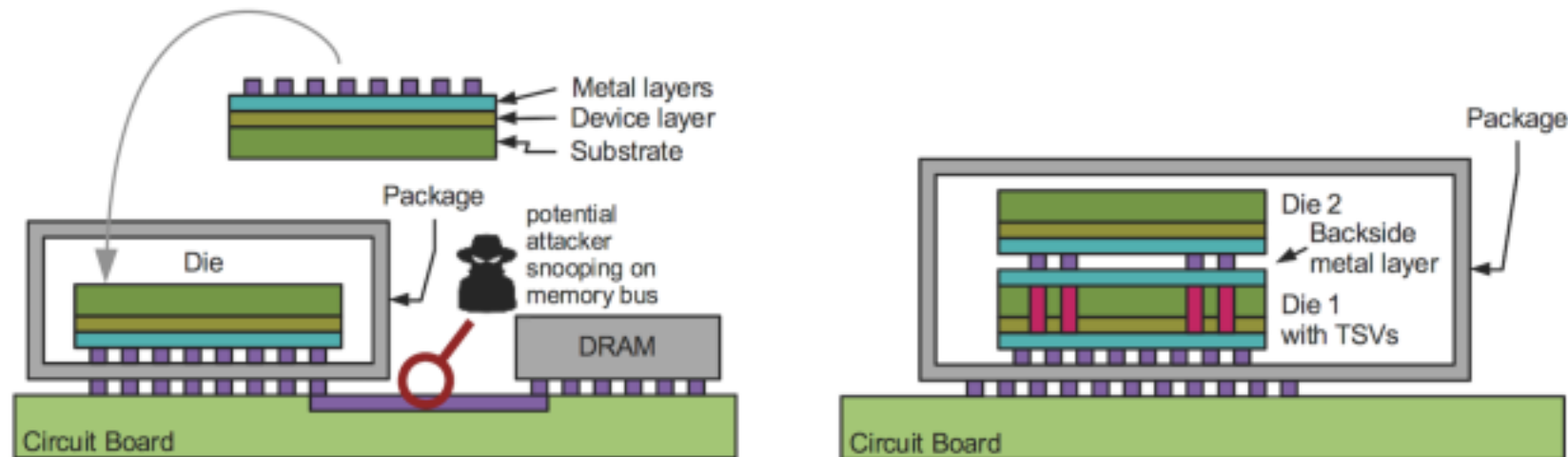


Leveraging 2.5D and 3D Integration



With 2.5D and 3D integration, the memory is brought into the same package as the main processor chip. Further, with embedded DRAM (eDRAM) the memory is on the same chip.

- Potentially probing attacks are more difficult
- Still limited memory (eDRAM around 128MB in 2017)



Security of Non-Volatile Memories and NVRAMs



- Non-volatile memories (NVMs) can store data even when there is no power
- Non-volatile random-access memory (NVRAM) is a specific type of NVM that is suitable to serve as a computer system's main memory, and replace or augment DRAM
- Many types of NVRAMs:
 - ReRAM – based on memristors, stores data in resistance of a dielectric material
 - FeRAM – uses ferroelectric material instead of a dielectric material
 - MRAM – uses ferromagnetic materials and stores data in resistance of a storage cell
 - PCM – typically uses chalcogenide glass where different glass phases have different resistances

Security considerations

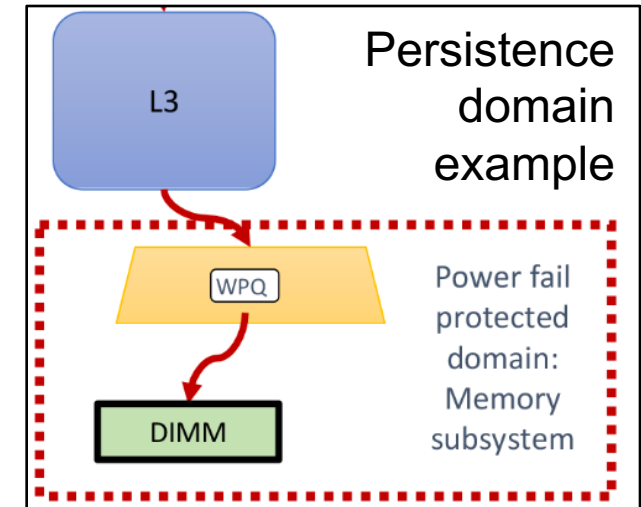
- Data remanence makes passive attacks easier (e.g. data extraction)
- Data is maintained after reboot or crash (security state also needs to be correctly restored after reboot or crash)

Features of Systems using NVRAMs



Persistence:

- Data persists across reboots and crashes, possibly with errors
- Need atomicity for data larger than one memory word (either all data or no data is “persisted”)
 - E.g. Write Pending Queue (WPQ) – memory controller has non-volatile storage or enough stored charge to write pending data back to the NV-DIMM or NVRAM



Granularity of persistence:

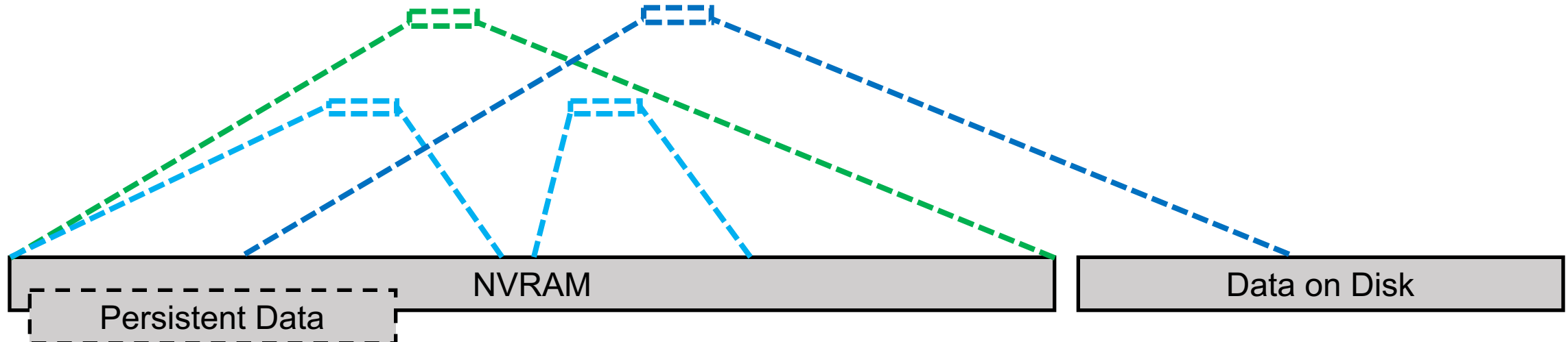
- Hide non-volatility from the system: simply use memory as DRAM replacement
- Expose non-volatility to the system: allow users to select which data is non-volatile
 - Linux support through Direct Access (DAX) since about 2014
 - Developed for NV-DIMMs (e.g., battery backed DRAM, but works for NVRAMs)

Image: https://www.usenix.org/system/files/login/articles/login_summer17_07_rudoff.pdf

Integrity Protection of NVRAMs



- For integrity, the integrity tree needs to additionally consider:
 - Atomicity of memory updates for data and related security state (so it is correct after reboot or a crash)
 - Which data in NVRAM is to be persisted (i.e. granularity)



Encrypted, Hashed, Oblivious Access Memory Assumption



Off-chip memory is untrusted and the contents is assumed to be protected from the snooping, spoofing, splicing, replay, and disturbance attacks:

- **Encryption** – snooping and spoofing protection
- **Hashing** – spoofing, splicing, replay (counters must be used), and disturbance protection
- **Oblivious Access** – snooping protection

Secure Processor Architectures
Trusted Execution Environments
Hardware Roots of Trust
Memory Protections

Multiprocessor and Many-core Protections

Side-Channels Threats and Protections
Principles of Secure Processor Architecture Design



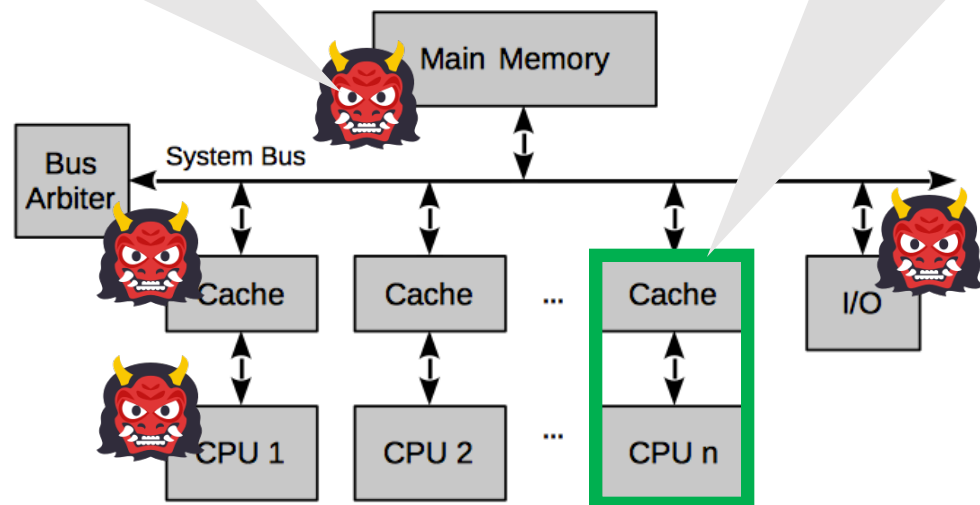
Multiprocessor Architectures



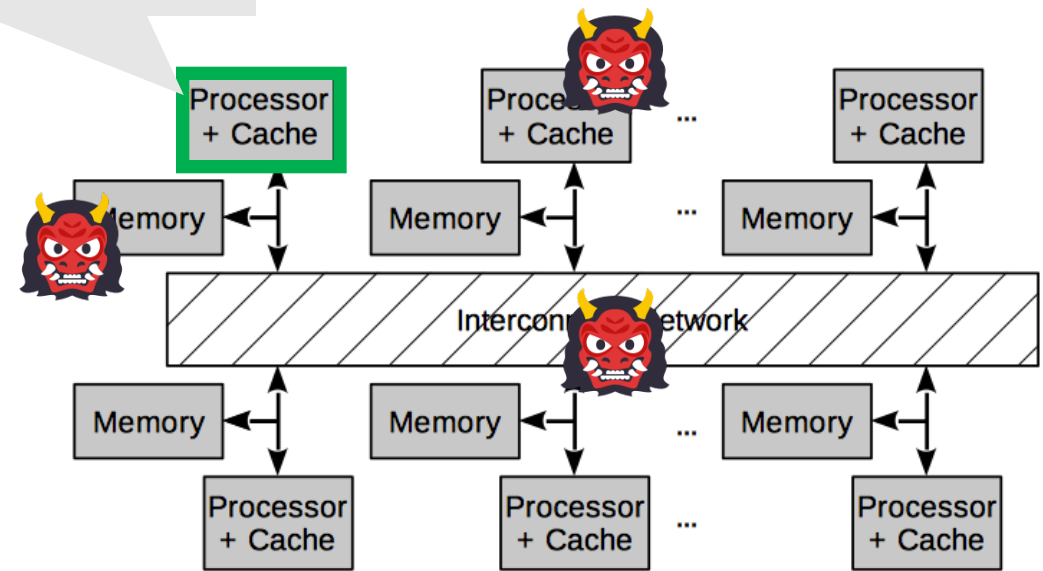
Symmetric Multi Processing (SMP) and Distributed Share Memory (DSM) also referred to as Non-Uniform Memory Access (NUMA) offer two ways of connecting many CPUs together.

Other components on the same system are untrusted

Individual processors are still trusted



SMP



DSM / NUMA

Emoji Image:
<https://www.emojione.com/emoji/1f479>

SMP Protections



Encrypt traffic on the bus between processors

- Each source-destination pair can share a hard-coded key
- Or use distribute keys using public key infrastructure (within a computer)

Use MACs for integrity of messages

- Again, each source-destination pair can share a key

Use Merkle trees for memory protection

- Can snoop on the shared memory bus to update the tree root node as other processors are doing memory accesses
- Or per-processor tree

DSM / NUMA Protections



Encrypt traffic on the bus between processors

- Again need a shared key

Use MACs for integrity of messages

- Again, each source-destination pair can share a key

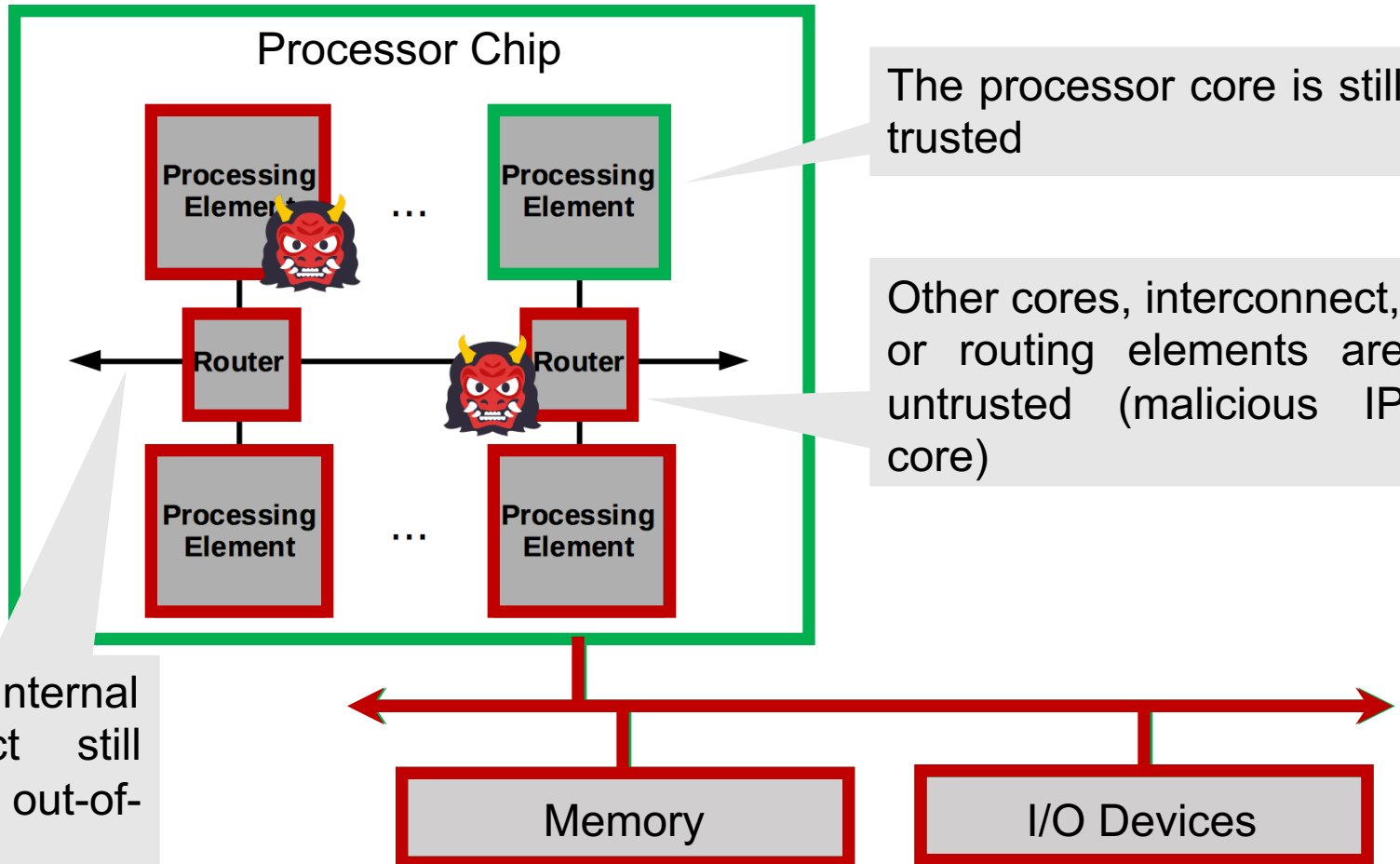
Use Merkle trees for memory protection

- No-longer can snoop on the traffic (DSM is point to point usually)

Many-core Trust Boundary



Trusted processor chip boundary is reduced in most research focusing on many-core security



The processor core is still trusted

Other cores, interconnect, or routing elements are untrusted (malicious IP core)

Probing of internal interconnect still assumed out-of-scope.

Architecture and Hardware Security Intersection



With many-core chips, the threats architects worry about start to overlap with hardware security researchers' work

- Untrusted 3rd party intellectual property (IP) cores
- Malicious foundry
- Untrusted supply chain

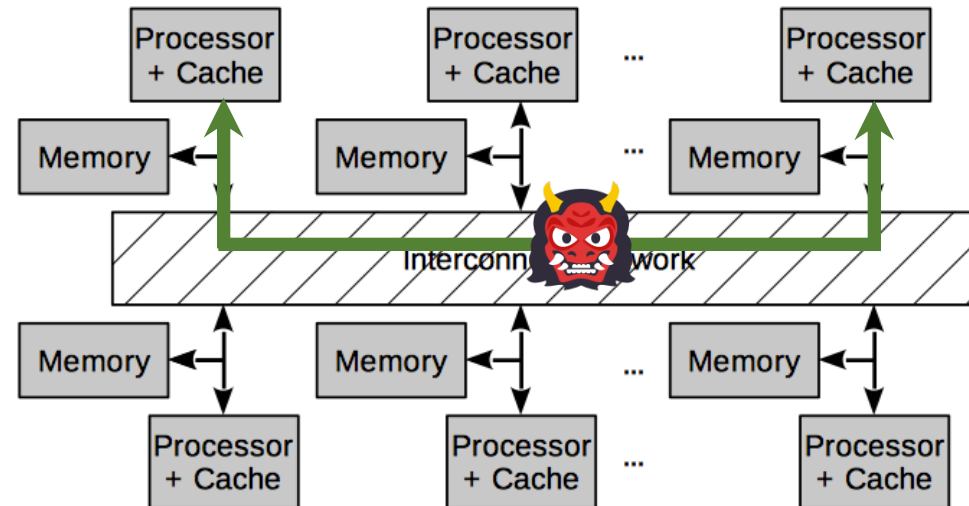
Architecture solutions (add encryption, add hashing, etc.) complement defenses developed by hardware security experts (split manufacturing, etc.).

Protected Inter-processor Communication Assumption



In addition to the existing assumption about protected memory communication, designs with multiple processors or cores assume the inter-processor communication will be protected:

- Confidentiality
- Integrity
- Communication pattern protection



Emoji Image:
<https://www.emojione.com/emoji/1f479>

Performance Challenges



Interconnects between processors are very fast:

- E.g. HyperTransport specifies speeds in excess of 50 GB/s
 - AES block size is 128 bits
 - Encryption would need 3 billion (giga) AES block encryptions or decryptions per second
- Tricks such as counter mode encryption can help
 - Only XOR data with a pad
 - But need to have or predict counters and generate the pads in time

Secure Processor Architectures

Trusted Execution Environments

Hardware Roots of Trust

Memory Protections

Multiprocessor and Many-core Protections

15:00 – 15:10 Break

Side Channel Threats and Protections

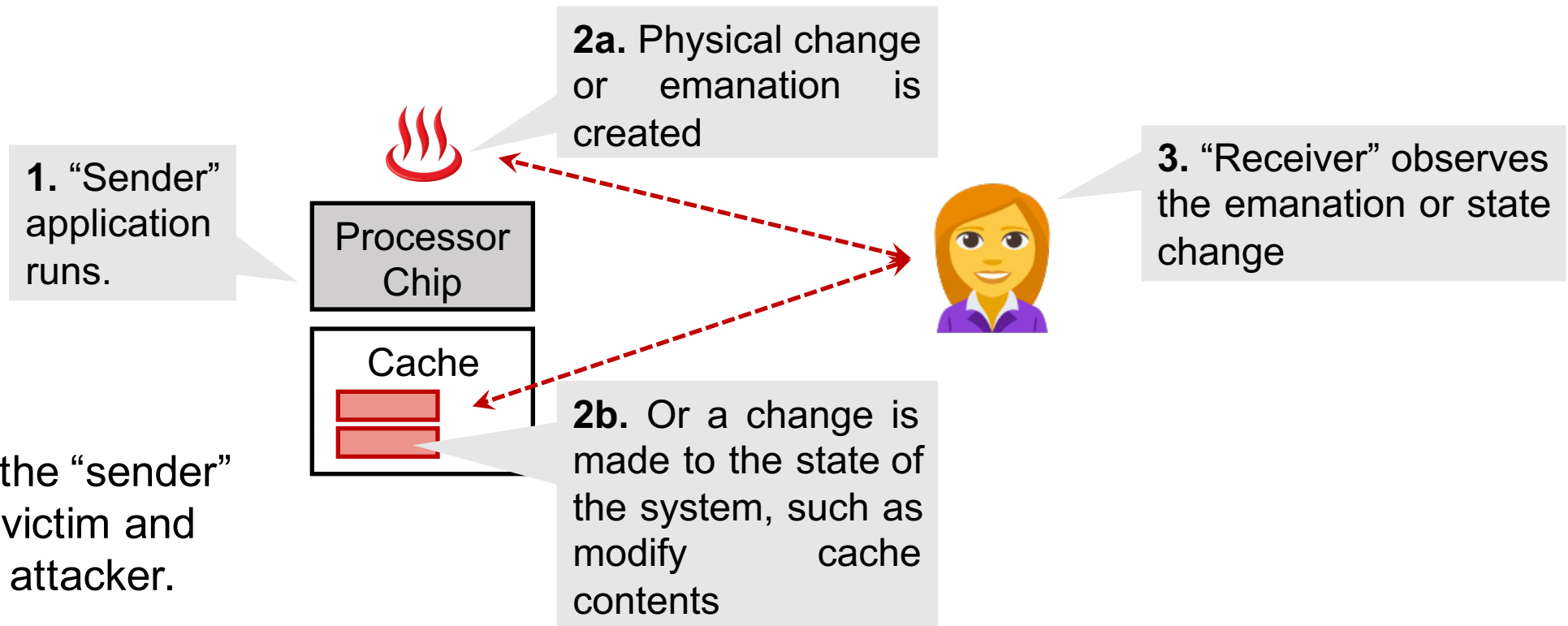
Principles of Secure Processor Architecture Design



Side and Covert Channels



A **covert channel** is an intentional communication between a sender and a receiver via a medium not designed to be a communication channel.



In a **side channel**, the "sender" is an unsuspecting victim and the "receiver" is the attacker.

Covert Channels



Covert Channel – a communication channel that was not intended or designed to transfer information, typically leverage unusual methods for communication of information, never intended by the system’s designers

- Timing
- Power
- Thermal emanations
- Electro-magnetic (EM) emanations
- Acoustic emanations

Covert channel is easier to establish, a precursor to side-channel attack

Side Channels



Side Channel – is similar to a covert channel, but the sender does not intend to communicate information to the receiver, rather sending (i.e. leaking) of information is a side effect of the implementation and the way the computer hardware or software is used.

- Timing
- Power
- Thermal emanations
- Electro-magnetic (EM) emanations
- Acoustic emanations

Differentiate side channel from covert channel depending on who controls the “sender”

Side Channels – Victim to Attacker



Typically a side channel is **from an unsuspecting victim to an attacker.**

- Goal is to extract some information from victim
- Victim does not observe any execution behavior change



Side Channels – Attacker to Victim



A side channel can also exist **from attacker to victim**.

- Attacker's behavior can "send" some information to the victim
- The information, in form of processor state for example, affects how the victim behaves unbeknownst to them

E.g. modulate branch predictor state to affect execution of the victim

Victim's operation depends on the information sent from attacker

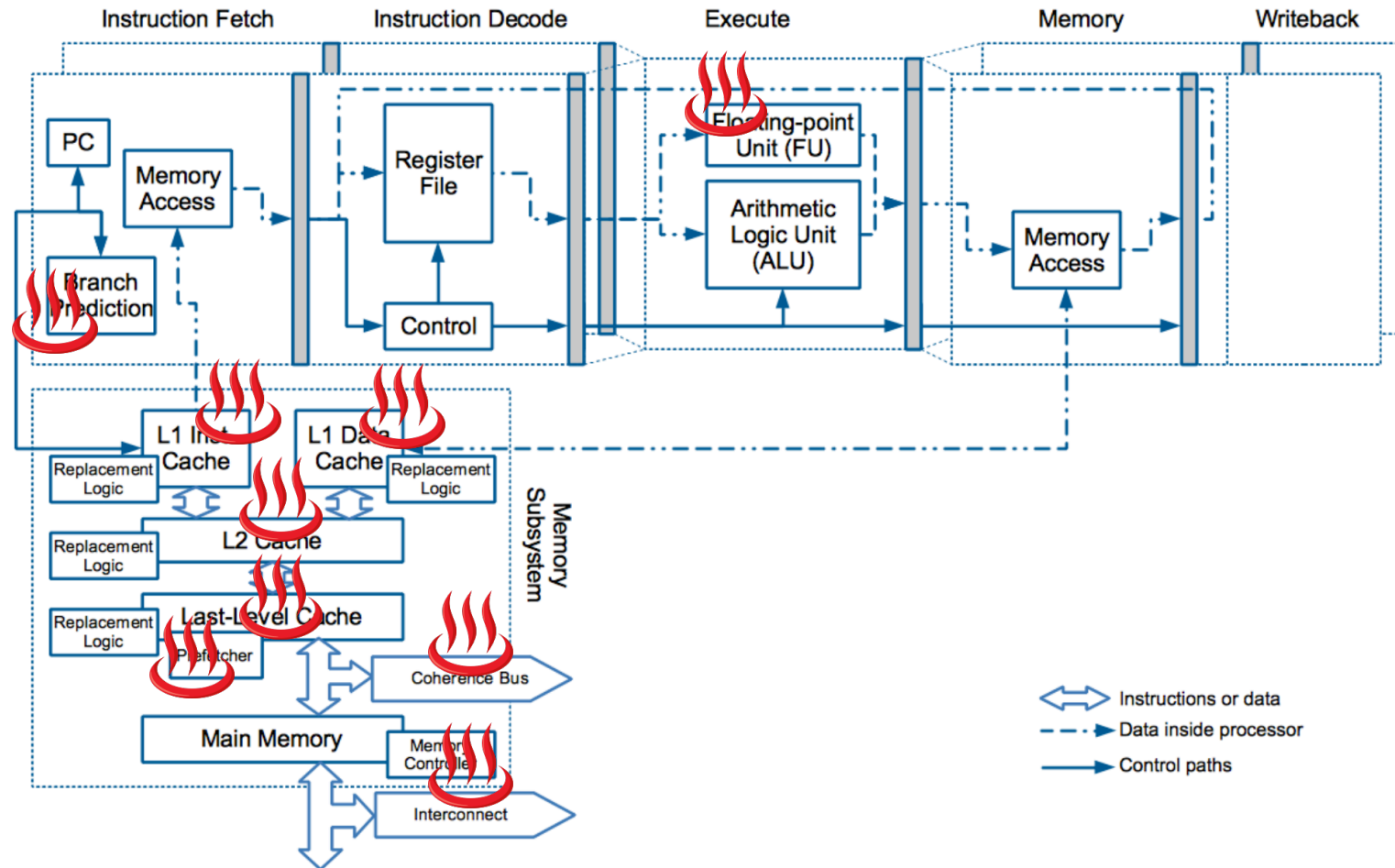


Attacker modulates some information that is sent to victim

Timing Side Channels Inside a Processor



Many components of a modern processor pipeline can contribute to side channels.



Sources of Timing Side Channels



Five source of side channels that can lead to attacks

1. **Variable Instruction Execution Timing** – Execution of different instructions takes different amount of time
2. **Functional Unit Contention** – Sharing of hardware leads to contention, whether a program can use some hardware leaks information about other programs
3. **Stateful Functional Units** – Program's behavior can affect state of the functional units, and other programs can observe the output (which depends on the state)
4. **Memory Hierarchy** – Data caching creates fast and slow execution paths, leading to timing differences depending on whether data is in the cache or not
5. **Physical Emanations** – Execution of programs affects physical characteristics of the chip, such as thermal changes, which can be observed

Variable Instruction Execution Timing



Computer architecture principles of **pipelining** and **making common case fast** drive processor designs where certain operations take more time than others – program execution timing may reveal which instruction was used.

- Multi-cycle floating point vs. single cycle addition
- Memory access hitting in the cache vs. memory access going to DRAM

Constant time software implementations can choose instructions to try to make software run in constant time

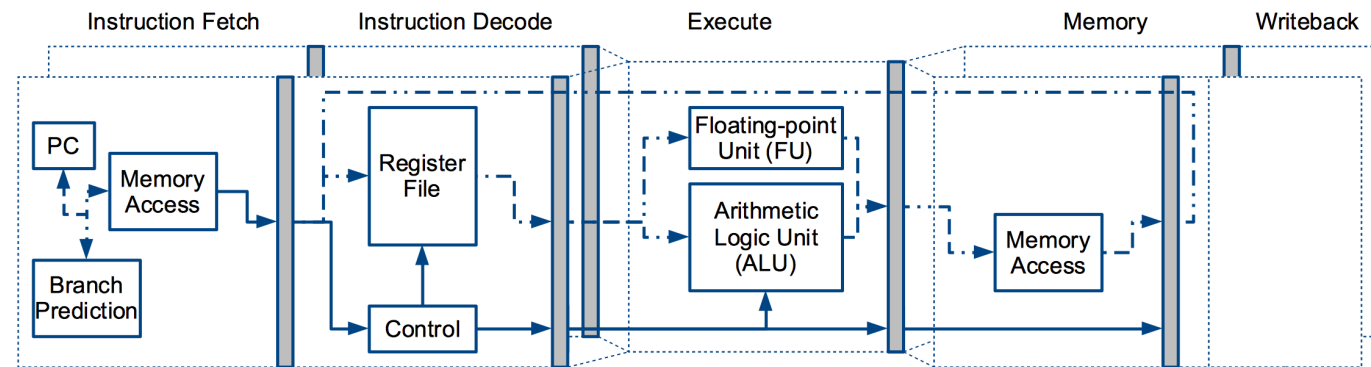
- Arithmetic is easiest to deal with
- Caches may need to be flushed to get constant memory instruction timing
- No way to flush state of functional units such as branch predictor

Functional Unit Contention



Functional units within processor are re-used or shared to save on area and cost of the processor resulting in varying program execution.

- Contention for functional units causes execution time differences



Spatial or Temporal Multiplexing allows to dedicate part of the processor for exclusive use by an application

- Negative performance impact or need to duplicate hardware

Stateful Functional Units



Many functional units inside the processor keep some history of past execution and use the information for prediction purposes.

- Execution time or other output may depend on the state of the functional unit
- If functional unit is shared, other programs can guess the state (and thus the history)
- E.g. caches, branch predictor, prefetcher, etc.

Flushing state can erase the history.

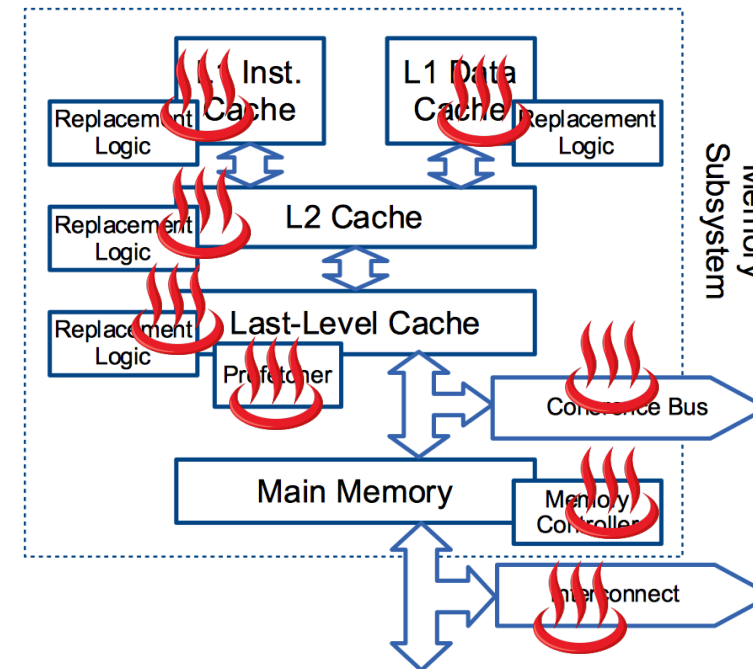
- Not really supported today
- Will have negative performance impact

Timing Side Channels in Memory Hierarchy



Memory hierarchy aims to improve system performance by hiding memory access latency (creating fast and slow executions paths); and parts of the hierarchy area a shared resource.

- Cache replacement logic
 - Inclusive caches
 - Non-inclusive caches
 - Exclusive caches
- Prefetcher logic
 - Also speculative instruction fetching from processor core
- Memory controller
- Interconnect
- Coherence bus



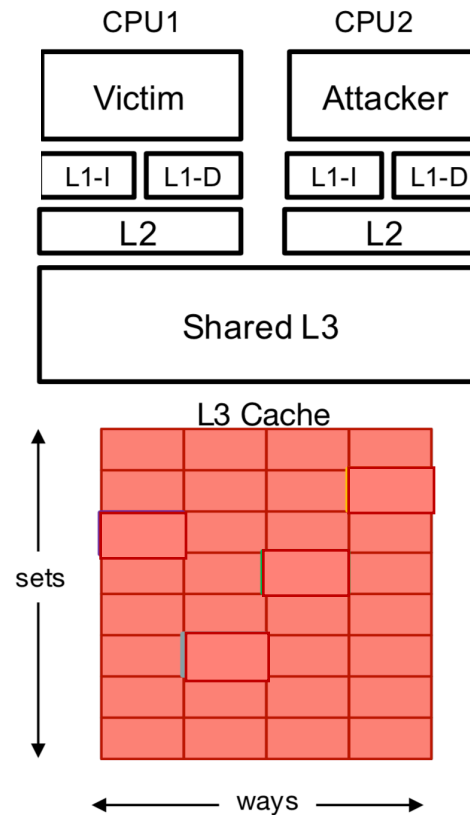
Emoji Image:
<https://www.emojione.com/emoji/2668>

Timing Cache Side Channels



Sharing of cache between two programs can let attacker program learn some information about a victim program based on observed timing of cache hits and misses.

E.g. Prime+Probe attack



Timing Side Channels due to Other Components



- **Prefetcher** – is used to prefetch data that may be used in future
 - Speculative Execution – data is fetched if an instruction is executed speculatively
- **TLB** – translation look aside buffer is another type of cache
 - Page Walk Cache (PWC) in Intel processors, is a buffer inside TLB
- **Memory Controller** – controls the memory accesses and arbitrates between different cores or caches accessing the memory
- **Interconnect** – interconnect between different components within the chip
- **Coherence bus** – interconnect between the chip and other chips or memory

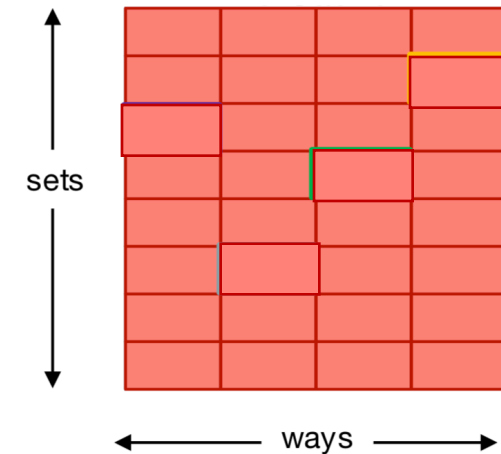
Meltdown



Meltdown vulnerability can be used to break isolation between user applications and the operating system.

1. Attempt to read data from kernel memory (mapped into address space of application)
2. Before an exception is raised, following instructions are speculatively executed
3. Exception is raised, however...
4. Cache state is modified
5. Processor cleans up the state, **but** data is left in cache

```
raise_exception();  
access(probe_array[data * 4096]);
```



Meltdown



Meltdown combines multiple attacks:

- Out-of-order execution causes permission checks to be done after operation already executes (only affects some processors)
- Cache state is not cleaned up, so one application can observe what the other did

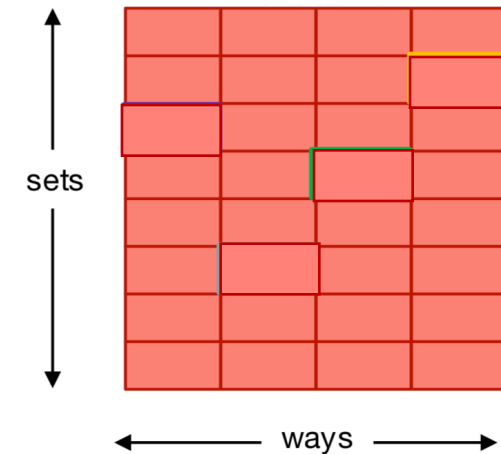
Spectre



Spectre vulnerability can be used to break isolation between different applications.

1. Attacker “trains” branch predictor
2. If statement in example is executed (predicted true)
3. Secret data from array1 is used as index to array2
4. Cache state is modified
5. Branch is resolved, processor cleans up the state, **but** data is left in cache

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```



Spectre



Spectre combines multiple attacks:

- Branch predictor state is not cleaned up, so one application can affect another
- Cache state is not cleaned up, so one application can observe what the other did

Foreshadow



Foreshadow vulnerability is similar to Spectre, but targets Intel SGX.

- Attack allows for speculative access to protected data in SGX memory
 - Data is encrypted in DRAM
 - But data is unencrypted in caches
- If the protected data is loaded into L1 cache by the victim (SGX enclave), attacker may be able to speculatively access it before processor determines that the access is forbidden.
- Difficult to exploit for true attack due to timing and data having to be in L1 cache

Classical vs. Speculative Side-Channels



Side channels can now be classified into two categories:

- **Classical** – which do not require speculative execution
- **Speculative** – which are based on speculative execution

Difference is victim is not fully in control of instructions they execute (i.e. some instructions are executed speculatively)

State of functional unit is modified by victim and it can be observed by the attacker via timing changes

Root cause of the attacks remains the same

Defending classical attacks defends speculative attacks as well, but not the other way around

Focusing only on speculative attacks does not mean classical attacks are prevented, e.g. defenses for cache-based attacks

Speculation Window



Key concept for speculative side-channel attacks is the speculation window

Speculation window:

- Amount of time from when a speculatively executed instructions start to issue, until when the instruction is squashed or becomes non-speculative
- Whole attack has to fit into speculation window
 - E.g. cache Flush+Reload attack requires to fetch data from main memory, thus window has to be bigger than about 300 cycles
 - E.g. Foreshadow attack requires fetch from L1 cache, so few cycles window is enough

Cache and Memory Access Latencies

<i>L1</i>	<i>1 cycle</i>
<i>L2</i>	<i>10 cycles</i>
<i>L3</i>	<i>50 cycles</i>
<i>Memory</i>	<i>200~300 cycles</i>

Side Channels due to Physical Emanations



Side-channels can be also observed from outside of the computer system, notably through physical emanations.

- Thermal

Require measuring temperature. Thermal channels possible in data centers without physical presence.

- Electromagnetic

Require measuring EM radiation. Today need dedicated equipment.

- Acoustic

Require measuring sound. Today need dedicated equipment.

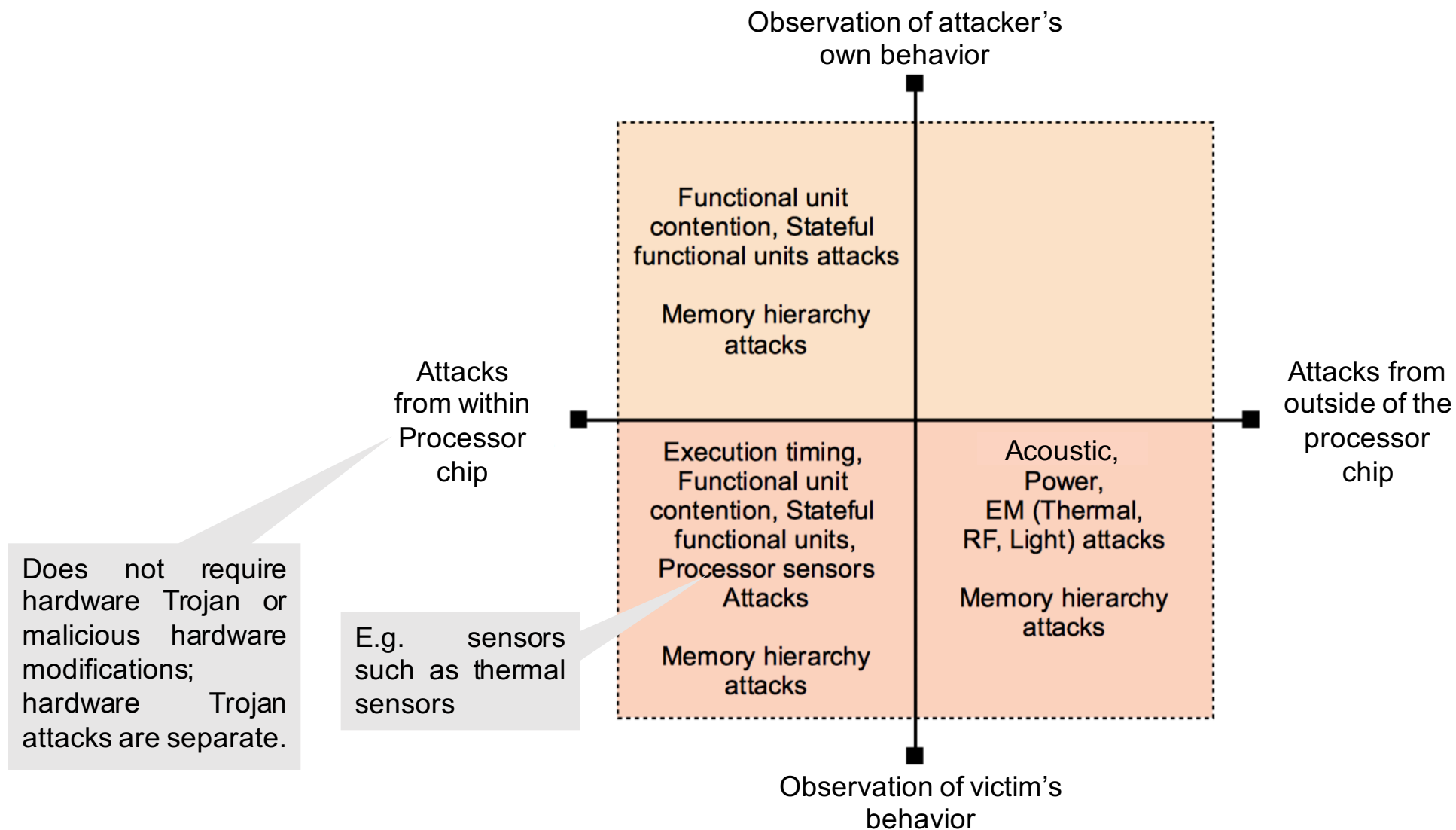
Timing Side Channel Bandwidths



The Orange Book, also called the Trusted Computer System Evaluation Criteria (TCSEC), specifies that a channel bandwidth exceeding a rate of **100 bps** is a **high bandwidth channel**.



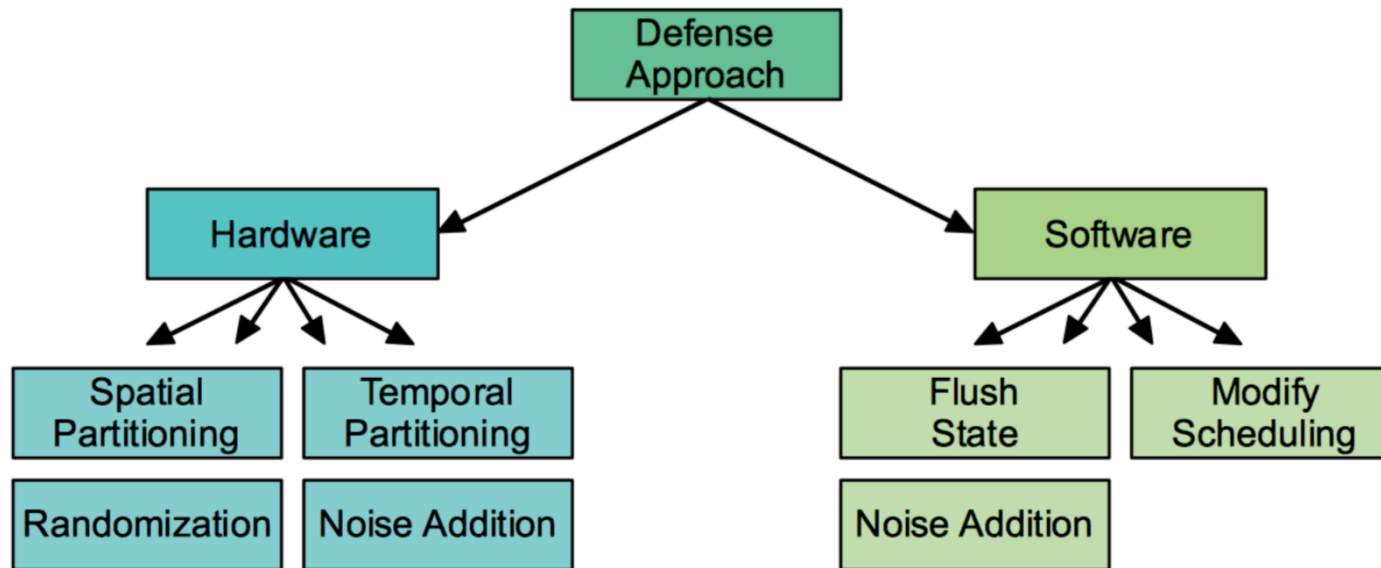
Side Channel Classification



Timing Channel Defense Strategies



Hardware and software based defenses are possible. Most will result in performance degradation.



Secure Caches to Defend Side Channels



Numerous academic proposals have presented different secure cache architectures that aim to defend against different cache-based side channels.

Approximate evaluation of 10 secure cache proposals:

	PL Cache	SecVerilog Cache	RP Cache	Newcache	Random Fill Cache	Sanctum Cache	SecDCP Cache	SP Cache	SHARP Cache	NoMo Cache
Confidentiality	×	×	×	✓	×	✓	×	✓	×	×
Integrity	×	×	×	✓	×	✓	×	✓	×	×
a. Access Contention Attack	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
b. Access Reuse Attack	×	✓	✓	✓	✓	—	✓	—	✓	×
c. Timing Contention Attack	✓	✓	✓	✓	×	✓	✓	✓	✓	✓
d. Timing Reuse Attack	×	×	×	×	✓	×	×	×	×	×

Partitioning and **randomization** are most effective techniques used in these caches

Example: Intel's Side Channel Defenses



Intel's Resource Director Technology (RDT) provides the hardware framework to monitor and manage shared CPU resources, like cache and memory bandwidth.

- Cache Monitoring Technology (CMT)
- Memory Bandwidth Monitoring (MBM)
- Cache Allocation Technology (CAT)
- Code and Data Prioritization (CDP)
- Memory Bandwidth Allocation (MBA)

Shared units inside the processor (e.g. branch predictor) so far not considered, but could be important to protect.

Side Channel Free TEE Assumption



The protected software assumes that the TEE is side channel free.

- TCB hardware and software should clean up processor state to remote any side channels
- Memory hierarchy should defend protected software from side channels

Protected software still needs to defend against **internal interference** channels

- Software's own memory accesses interfere with each other
- Best to write constant time software

Side Channels as Attack Detectors



Side channels can be used to detect or observe system operation.

- Measure timing, power, EM, etc. to detect unusual behavior
- Similar to using performance counters, but attacker doesn't know measurement is going on

Tension between **side channels as attack vectors vs. detection tools.**

- Side channels are mostly used for attack today

Industry Standards for Evaluating System's Security



Orange Book or the Trusted Computer System Evaluation Criteria (TCSEC)

- Replaced by Common Criteria
- Standard for assessing the effectiveness of a computer system's security controls

Common Criteria

- Standard for computer security certification

FIPS 140-2

- Standard defining security levels for cryptographic modules

Secure Processor Architectures

Trusted Execution Environments

Hardware Roots of Trust

Memory Protections

Multiprocessor and Many-core Protections

Side Channel Threats and Protections

Principles of Secure Processor Architecture Design



Principles of Computer Architecture



Traditional computer architecture has six principles regarding processor design:

- Caching E.g. caching frequently used data in a small but fast memory helps hide data access latencies.
- Pipelining E.g. break processing of an instruction into smaller chunks that can each be executed sequentially reduces critical path of logic and improves performance.
- Predicting E.g. predict control flow direction or data values before they are actually computed allows code to execute speculatively.
- Parallelizing E.g. processing multiple data in parallel allows for more computation to be done concurrently.
- Use of indirection E.g. virtual to physical mapping abstracts away physical details of the system.
- Specialization E.g. custom instructions use dedicated circuits to implement operations that otherwise would be slower using regular processor instructions.

What are principles for secure architectures?

Review of Secure Processor Assumptions



Assumptions and how they are broken:

- Trusted Processor Chip Assumption
 - Small TCB Assumption
 - Open TCB Assumption
 - No Side-Effects Assumption
 - Benign Protected Software Assumption
 - Trustworthy TCB Execution Assumption
 - Protected Root of Trust Assumption
 - Fresh Measurement Assumption
 - Encrypted, Hashed, Oblivious Access Memory Assumption
 - Protected Inter-processor Communication Assumption
 - Side Channel Free TEE Assumption
- Invasive attacks, hardware Trojans, supply chain attacks
- Code bloat, proprietary code running on embedded security processor
- State in functional units not cleaned up
- Malware hidden in TEE
- No means to monitor TCB execution
- Compromised manufacturer database
- TOC-TOU attacks and no continuous measurement
- Lack of encryption, hashing or ORAM due to performance issues
- Lack of side channel protections

Principles of Secure Processor Architecture Design



Four principles for secure processor architecture design based on existing designs and also on ideas about what ideal design should look like.

- 1. Protect Off-chip Communication and Memory**
- 2. Isolate Processor State between TEE Execution**
- 3. Allow TCB Introspection**
- 4. Authenticate and Continuously Monitor TEE**

Additional design suggestions:

- Avoid code bloat
- Minimize TCB
- Ensure hardware security (Trojan prevention, supply chain issues, etc.)
- Use formal verification

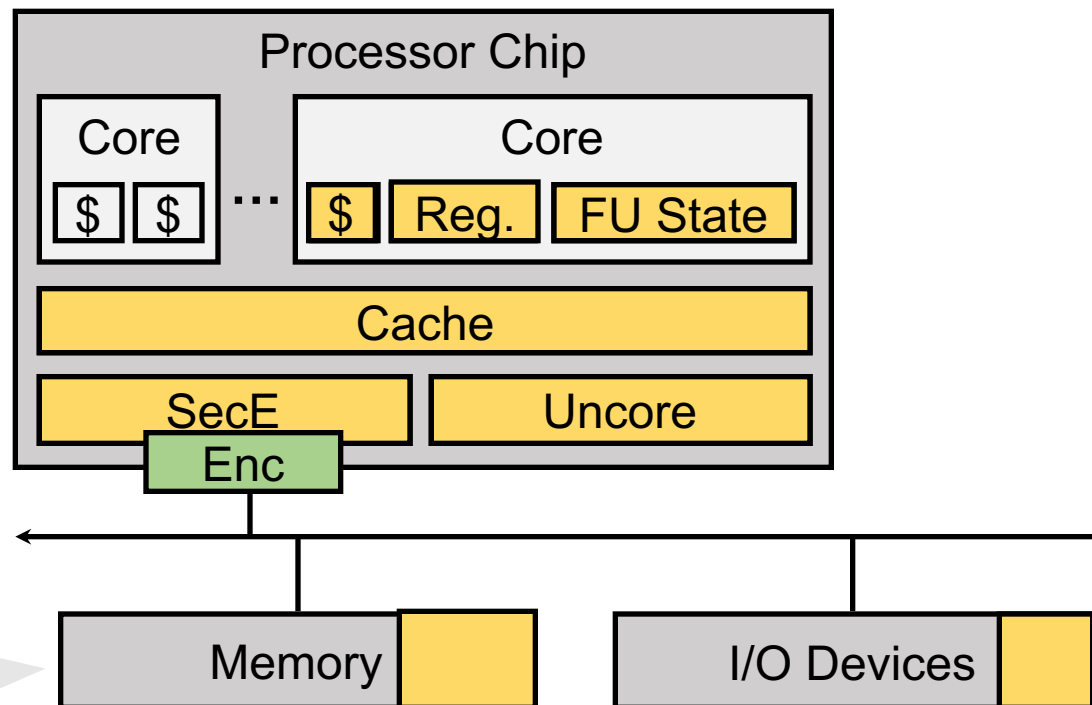
Protect Off-chip Communication and Memory



Off-chip components and communication are untrusted, need protection with **encryption**, **hashing**, **access pattern protection**.

Open research challenges:

- Performance
- Key distribution



E.g. encryption defends Cold boot style attacks on main memory.

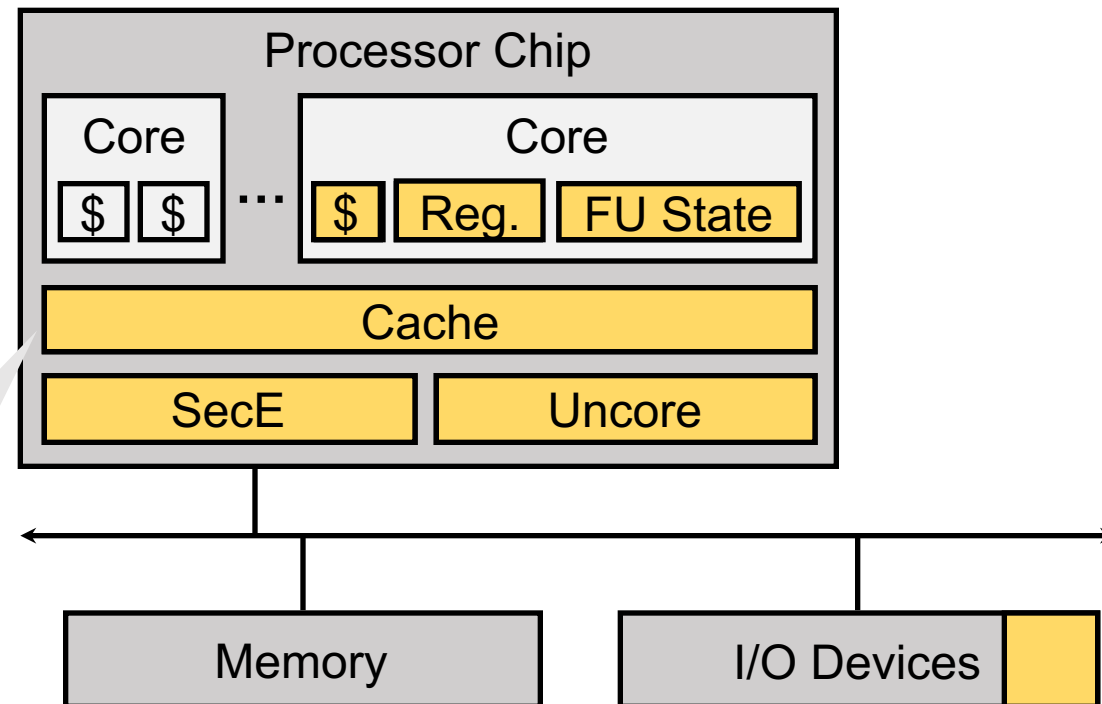
Isolate Processor State between TEE Execution



When switching between protected software, need to flush the state, or save and restore it, to prevent one software influencing another.

Open research challenges:

- Performance
- Finding all the state to flush or clean
- ISA interface to allow state flushing



E.g. flushing state defends Spectre and Meltdown type attacks.

Allow TCB Introspection

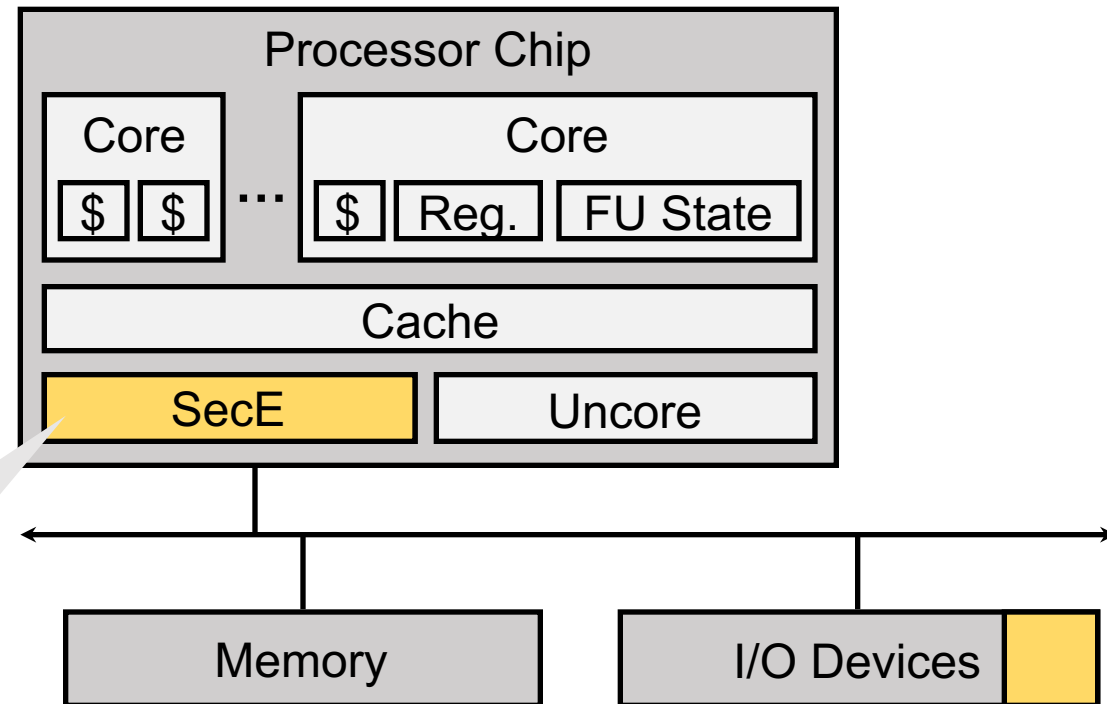


Need to ensure correct execution of TCB, through **open access to TCB design, monitoring, fingerprinting, and authentication.**

Open research challenges:

- ISA interface to introspect TCB
- Area, energy, performance costs due extra features for introspection
- Leaking information about TCB or TEE

E.g. open TCB design can minimize attacks on ME or PSP security engines



Authenticate and Continuously Monitor TEE

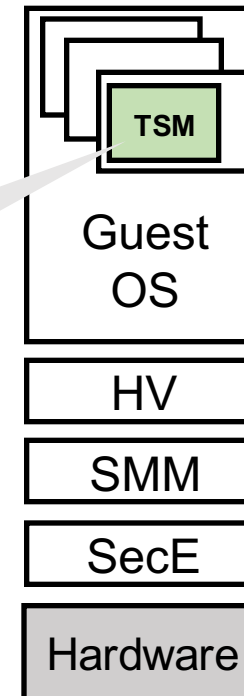


Monitoring of software running inside TEE, e.g. TSMs or Enclaves, gives assurances about the state of the protected software.

Open research challenges:

- Interface design for monitoring
- Leaking information about TEE

E.g. continuous monitoring of a TEE can help prevent TOC-TOU attacks.



Pitfalls and Fallacies



- Pitfall: Security by Obscurity

E.g. recent attacks on industry processors.
- Fallacy: Hardware Is Immutable

Most actually realized architectures use a security processor (e.g. ME or PSP).
- Pitfall: Wrong Threat Model

E.g. original SGX did not claim side channel protection, but researchers attacked it.
- Pitfall: Fixed Threat Model

Most designs are one-size-fits all solutions.
- Pitfall: Use of Outdated or Custom Crypto

E.g. today's devices will be in the field for many years, but do not use post-quantum crypto.
- Pitfall: Not Addressing Side Channels

Most architectures underestimate side channels.
- Pitfall: Requiring Zero-Overhead Security

Performance-, area-, or energy-only focused designs ignore security.
- Pitfall: Code Bloat

E.g. rather than partition a problem, large code pieces are ran instead TEEs; also TCB gets bigger and bigger leading to bugs.
- Pitfall: Incorrect Abstraction

Abstraction (e.g. ISA assumptions) does not match how device or hardware really behaves.

Pitfalls and Fallacies



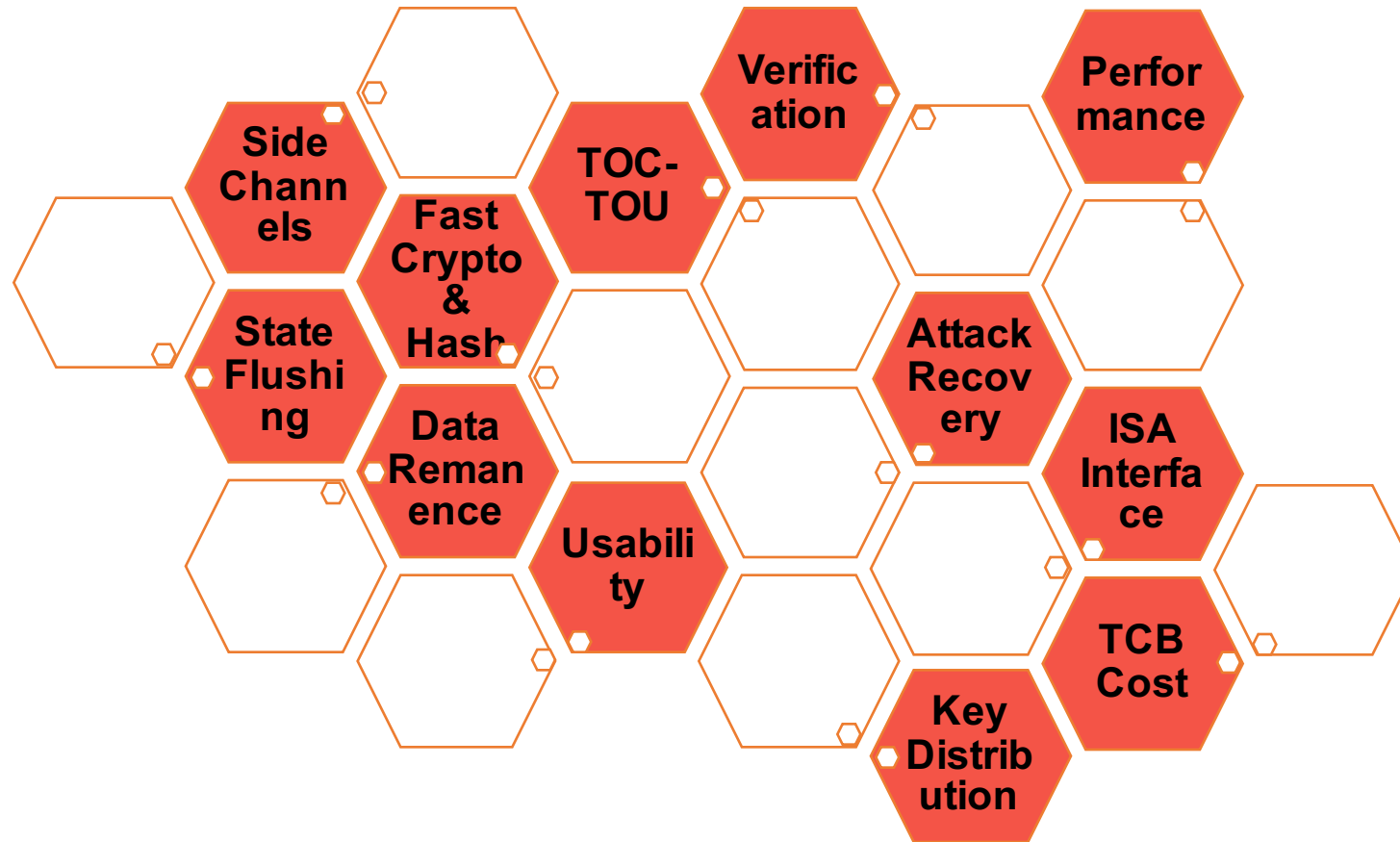
- Pitfall: Focus Only on Speculative Attacks
- ...

Defending only speculative attacks does not ensure classical attacks are also protected

Challenges in Secure Processor Design



A number of challenges remain in research on secure processor designs:

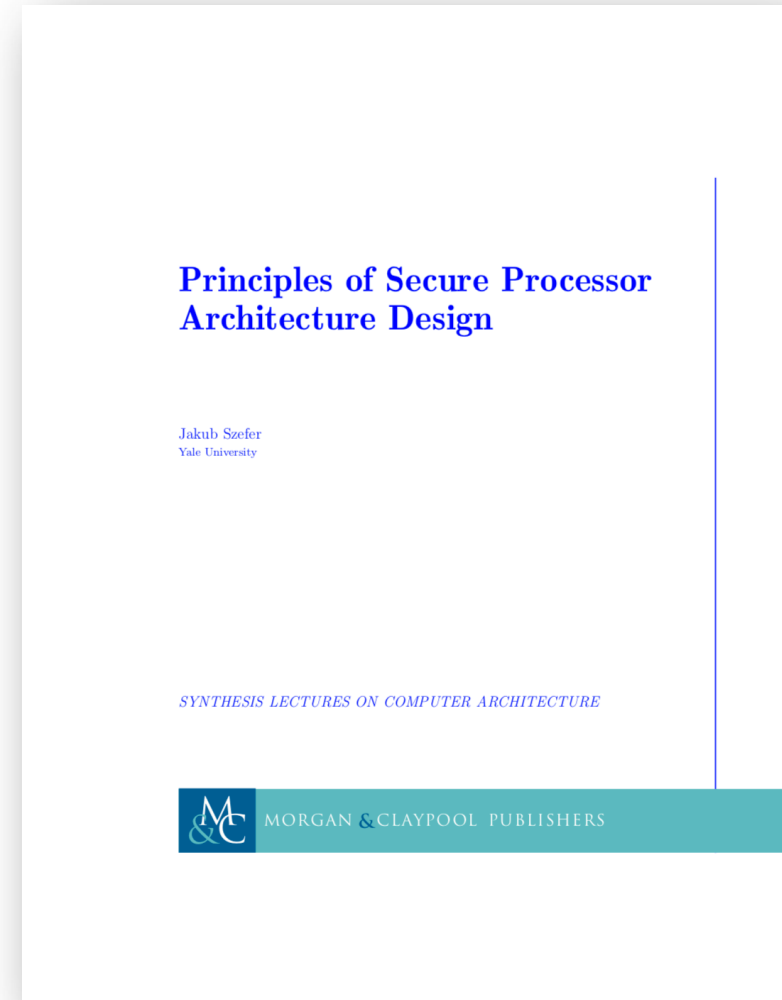


The Book



Jakub Szefer, "Principles of Secure Processor Architecture Design," in Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, October 2018.

<http://caslab.csl.yale.edu/books/>



Summer Course on Processor Architecture Security

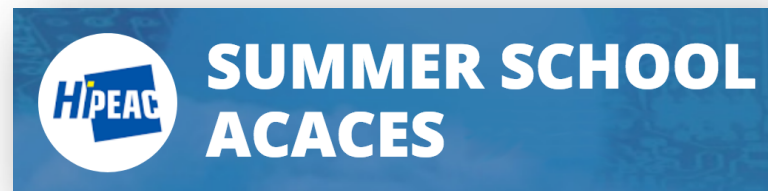


Who: Jakub Szefer

What: Summer Course on **Processor Architecture Security**

Where: at the 15th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), in Rome, Italy

When: Sunday evening July 14th, 2019 until Friday evening July 19th, 2019



Acknowledgement



Work on this tutorial was possible in part through support from NSF grants number **1716541**, **1524680**, and NSF CAREER award number **1651945**.

Presentation of past tutorials were made possible in part by **Yale University**.

Special thanks to students **Wenjie Xiong**, **Wen Wang**, **Shuwen Deng**, **Shanquan Tian**, and visiting student **Shuai Chen**, for presentation feedback.

And thanks to past tutorial participants for suggestions and their feedback on improving the slides.



Thank You!